PROGRAMANDO EM C/C++

"A Bíblia"

O Melhor Guia Para a Programação em C/C++

Kris Jamsa, Ph.D. / Lars Klander

Inclui CD-ROM
Borland Turbo C++ Lite – Tudo de que você precisa para criar programas C++ e muito mais!

MAKRON Books

MAKRON
Books

PROGRAMANDO EM C/C++

A BÍBLIA

KRIS JAMSA, PH.D.

LARS KLANDER

Tradução e Revisão Técnica

Jeremias René D. Pereira dos Santos

Analista de Software

MAKRON *Books* do Brasil Editora Ltda.
Rua Tabapuã, 1.348, Itaim-Bibi
CEP 04533-004 — São Paulo — SP
(011) 829-8604 e (011) 820-6622
makron@book.com.br

São Paulo • Rio de Janeiro • Ribeirão Preto • Belém • Belo Horizonte • Brasília • Campo Grande • Cuiabá • Curitiba • Florianópolis • Fortaleza • Goiânia • Manaus • Natal • Porto Alegre • Recife • Salvador

Barcelona • Bogotá • Buenos Aires • Caracas • Ciudad del México • Frankfurt • Guadalajara • Lisboa • Londres • Madrid • Montevideo • New York • Paris • Porto • Santiago

Do Original Jamsa's C/C++ Programmer's Bible

Copyright © 1998 by Jamsa Press
Copyright © 1999 MAKRON *Books* do Brasil Editora Ltda.

Todos os direitos para a língua portuguesa reservados pela MAKRON *Books* do Brasil Editora Ltda. Nenhuma parte desta publicação poderá ser reproduzida, guardada pelo sistema "retrieval" ou transmitida de qualquer modo ou por qualquer outro meio, seja este eletrônico, mecânico, de fotocópia, de gravação, ou outros, sem prévia autorização, por escrito, da Editora.

EDITOR: MILTON MIRA DE ASSUMPÇÃO FILHO

Gerente de Produção
Sílas Camargo

Editora Assistente
Eugênia Pessotti

Produtora Editorial
Salete Del Guerra

Design de Capa
Marianne Helm
James Rehrauer

Editoração Eletrônica: ERJ Informática Ltda.

Dados de Catalogação na Publicação

Programando em C/C++ — A Bíblia; Kris Jamsa e Lars Klander
tradução: e revisão técnica: Jeremias René D. Pereira dos Santos
São Paulo: MAKRON *Books*, 1999.

Título original: Jamsa's C/C++ Programmer's Bible

ISBN: 85.346.1025-8

SUMÁRIO

INTRODUÇÃO À LINGUAGEM C

Uma Introdução à Programação 1
Criando um Arquivo-Fonte em ASCII 2
Compilando Seu Programa 3
Compreendendo os Erros de Sintaxe 4
Estrutura de um Típico Programa em C 5
Acrescentando Comandos aos Seus Programas 6
Exibindo Saída em uma Nova Linha 7
Considera as Letras Maiúsculas e Minúsculas Diferentes 8
Compreendendo os Erros Lógicos 9
Compreendendo o Processo de Desenvolvimento de um Programa 10
Compreendendo os Tipos de Arquivo 11
Compreendendo Melhor o Linkeditor 12
Compreendendo os Arquivos de Cabeçalho 13
Ajudando o Compilador a Localizar os Arquivos de Cabeçalho ... 14
Agilizando as Compilações 15
Comentando Seus Programas 16
Melhorando a Legibilidade do Seu Programa 17
Prestando Atenção às Mensagens de Advertência do Compilador . 18
Controlando as Advertências do Compilador 19
Usando Comentários para Excluir Comandos do Programa 20
Compreendendo a Importância dos Nomes 21
Compreendendo o Ponto-e-Vírgula 22
Apresentando as Variáveis 23
Atribuindo um Valor a uma Variável 24
Compreendendo os Tipos de Variáveis 25
Declarando Múltiplas Variáveis do Mesmo Tipo 26
Comentando Suas Variáveis na Declaração 27
Atribuindo Valores às Variáveis na Declaração 28
Inicializando Múltiplas Variáveis durante a Declaração 29
Usando Nomes Representativos para as Variáveis 30
Compreendendo as Palavras-chave de C 31
Compreendendo as Variáveis do Tipo int 32
Compreendendo as Variáveis do Tipo char 33
Compreendendo as Variáveis do Tipo float 34
Compreendendo as Variáveis do Tipo double 35
Atribuindo Valores aos Valores em Ponto Flutuante 36
Compreendendo os Modificadores de Tipo 37
Compreendendo o Modificador de Tipo unsigned 38
Compreendendo o Modificador de Tipo long 39
Combinando os Modificadores de Tipo unsigned e long 40
Trabalhando com os Valores Grandes 41
Compreendendo o Modificador de Tipo register 42
Compreendendo o Modificador de Tipo short 43
Omitindo int das Declarações Modificadas 44
Compreendendo o Modificador de Tipo signed 45
Múltiplas Operações de Atribuição 46
Atribuindo o Valor de um Tipo de Variável a um Tipo Diferente 47
Criando Seus Próprios Tipos 48
Atribuindo um Valor Hexadecimal ou Octal 49
Compreendendo o Extravasamento 50
Compreendendo a Precisão 51
Atribuindo Apóstrofos e Outros Caracteres 52
Introdução ao Comando printf 53
Exibindo Valores do Tipo int Usando printf 54
Imprimindo um Valor Inteiro Octal ou Hexadecimal 55
Exibindo Valores do Tipo unsigned int Usando printf 56
Exibindo Valores do Tipo long int Usando printf 57
Exibindo Valores do Tipo float Usando printf 58
Exibindo Valores do Tipo char Usando printf 59
Exibindo Valores de Ponto Flutuante em um Formato Exponencial 60
Exibindo Valores em Ponto Flutuante 61
Exibindo uma String de Caracteres Usando printf 62
Exibindo um Endereço de Ponteiro Usando printf 63
Precedendo um Valor com um Sinal de Adição ou de Subtração 64
Formatando um Valor Inteiro Usando printf 65
Saída de Inteiros Preenchida com Zeros 66
Exibindo um Prefixo Antes dos Valores Octais ou Hexadecimais 67
Formatando um Valor em Ponto Flutuante Usando printf 68
Formatando a Saída Exponencial 69
Justificando à Esquerda a Saída de printf 70
Combinando os Especificadores de Formato de printf 71
Quebrando uma String de Caracteres em Duas Linhas 72
Exibindo Strings do Tipo near e far 73
Trabalhando com os Caracteres Escape de printf 74
Determinando o Número de Caracteres Que printf Exibiu 75
Usando o Valor de Retorno de printf 76
Usando o Controlador de Dispositivo ANSI 77
Usando o Controlador ANSI para Limpar Sua Tela 78
Usando o Controlador ANSI para Exibir as Cores da Tela 79
Usando o Controlador ANSI para Posicionar o Cursor 80
Executando Operações Matemáticas Básicas em C 81
Compreendendo a Aritmética do Módulo 82
Compreendendo a Precedência e a Associatividade dos Operadores 83
Forçando a Ordem de Avaliação dos Operadores 84
Compreendendo o Operador de Incremento de C 85
Compreendendo o Operador de Decremento de C 86
Compreendendo uma Operação OU Bit a Bit 87
Compreendendo uma Operação E Bit a Bit 88
Compreendendo uma Operação OU Exclusivo Bit a Bit 89
Compreendendo a Operação Inverso Bit a Bit 90
Aplicando uma Operação ao Valor de uma Variável 91
Compreendendo o Operador Condicional de C 92
Compreendendo o Operador sizeof de C 93
Efetuando um Deslocamento Bit a Bit 94
Efetuando uma Rotação Bit a Bit 95
Compreendendo os Operadores Condicionais 96
Compreendendo o Processamento Iterativo 97
Compreendendo Como C Representa Verdadeiro e Falso 98
Testando uma Condição com if 99
Compreendendo os Comandos Simples e Compostos 100
Testando a Igualdade 101
Efetuando Testes Relacionais 102
Executando uma Operação Lógica E para Testar Duas Condições 103
Executando uma Operação Lógica OU para Testar Duas Condições 104
Executando uma Operação Lógica NÃO (NOT) 105
Atribuindo o Resultado de uma Condição 106
Declarando Variáveis Dentro de Comandos Compostos 107
Usando Recuos para Melhorar a Legibilidade 108
Usando a Verificação Estendida de Ctrl+Break 109
Testando Valores de Ponto Flutuante 110
Laço de Repetição Infinito 111
Testando uma Atribuição 112
Cuidado com os Comandos if-if-else 113
Executando Comandos um Número Específico de Vezes 114
Partes do Comando for São Opcionais 115
Decrementando os Valores em um Comando for 116
Controlando o Incremento do Laço for 117
Usando Laços for com Valores char e float 118

Compreendendo um Laço Null 119
Compreendendo o Laço Infinito 120
Usando o Operador Vírgula de C Dentro de um Laço for 121
Evite Modificar o Valor de uma Variável de Controle em um Laço for 122
Repetindo um ou Mais Comandos Usando um Laço while 123
Compreendendo as Partes de um Laço while 124
Repetindo um ou Mais Comandos Usando do 125
Compreendendo o Comando continue de C 126
Finalizando um Laço Usando o Comando break de C 127
Desvios com o Comando goto 128
Testando Múltiplas Condições 129
Compreendendo break Dentro de switch 130
Usando o Case default do Comando switch 131
Definindo Constantes no Seu Programa 132
Compreendendo a Expansão de Macros e de Constantes 133
Nomeando as Constantes e as Macros 134
Usando a Constante de Pré-processador _FILE_ 135
Usando a Constante de Pré-processador __LINE__ 136
Alterando a Contagem de Linha do Pré-processador 137
Gerando um Erro Incondicional do Pré-processador 138
Outras Constantes do Pré-processador 139
Registrando a Data e a Hora do Pré-processador 140
Testando a Adesão ao Padrão ANSI 141
Testando C++ versus C 142
Anulando a Definição de uma Macro ou Constante 143
Comparando Macros e Funções 144
Compreendendo Pragmas do Compilador 145
Aprendendo sobre os Valores e Macros Predefinidos 146
Criando Seus Próprios Arquivos de Cabeçalho 147
Usando #include <nomearq.h> ou #include "nomearq.h" 148
Testando Se um Símbolo Está Definido 149
Efetuando o Processamento if-else 150
Efetuando um Teste de Condição de Pré-processador Mais Poderoso 151
Realizando Processamento if-else e else-if 152
Definindo Macros e Constantes Que Requerem Múltiplas Linhas 153
Criando Suas Próprias Macros 154
Não Coloque Ponto-e-Vírgula nas Definições de Macros 155
Criando Macros Min e Max 156
Criando Macros QUADRADO e CUBO 157
Cuidado com os Espaços nas Definições das Macros 158
Compreendendo Como Usar os Parênteses 159
As Macros Não Têm Tipo 160

COMPREENDENDO AS STRINGS

Visualizando uma String de C 161
Como o Compilador Representa uma String de Caracteres 162
Como C Armazena as Strings de Caracteres 163
Aprendendo Como 'A' Difere de "A" 164
Representando um Apóstrofo Dentro de uma Constante de String 165
Determinando o Tamanho de uma String 166
Usando a Função strlen 167
Copiando os Caracteres de uma String em Outra String 168
Anexando o Conteúdo de uma String em Outra 169
Anexando n Caracteres em uma String 170
Transformando uma String em Outra 171
Não Ultrapasse o Tamanho de uma String 172
Determinando Se Duas Strings São ou Não Iguais 173
Ignorando a Caixa ao Determinar Se as Strings São Iguais 174
Convertendo uma String de Caracteres para Maiúsculas ou Minúsculas 175
Obtendo a Primeira Ocorrência de um Caractere em uma String 176
Retornando um Índice à Primeira Ocorrência de uma String ... 177
Encontrando a Última Ocorrência de um Caractere em uma String 178
Retornando um Índice à Última Ocorrência de uma String 179
Trabalhando com Strings do Tipo far 180
Escrevendo Funções String para as Strings far 181
Contando o Número de Ocorrências de um Caractere em uma String 182
Invertendo o Conteúdo de uma String 183
Atribuindo um Caractere Específico a uma String Inteira 184
Comparando Duas Strings de Caracteres 185
Comparando os Primeiros n Caracteres de Duas Strings 186
Comparando Strings sem Considerar a Caixa 187
Convertendo a Representação em String de Caracteres de um Número 188
Duplicando o Conteúdo de uma String 189
Encontrando um Caractere da Primeira Ocorrência de um Determinado Conjunto 190
Localizando uma Substring Dentro de uma String 191
Contando o Número de Ocorrências da Substring 192
Obtendo um Índice para uma Substring 193
Obtendo a Ocorrência Mais à Direita de uma Substring 194
Exibindo uma String sem o Especificador de Formato %s 195
Removendo uma Substring de Dentro de uma String 196
Substituindo uma Substring por Outra 197
Convertendo uma Representação Numérica ASCII 198
Determinando Se um Caractere É Alfanumérico 199
Determinando Se um Caractere É uma Letra 200
Determinando Se um Caractere Contém um Valor ASCII 201
Determinando Se um Caractere É um Caractere de Controle ... 202
Determinando Se um Caractere É um Dígito 203
Determinando Se um Caractere É um Caractere Gráfico 204
Determinando Se um Caractere É Maiúsculo ou Minúsculo 205
Determinando Se um Caractere É Imprimível 206
Determinando Se um Caractere É um Sinal de Pontuação 207
Determinando Se um Caractere Contém Espaço em Branco 208
Determinando Se um Caractere É um Valor Hexadecimal 209
Convertendo um Caractere para Maiúscula 210
Convertendo um Caractere para Minúsculo 211
Trabalhando com Caracteres ASCII 212
Escrevendo Saída Formatada em uma Variável String 213
Lendo a Entrada de uma String de Caracteres 214
"Tokenizando" as Strings para Poupar Espaço 215
Inicializando uma String 216

FUNÇÕES

Compreendendo as Funções 217
Usando Variáveis Dentro das Funções 218
Compreendendo main Como uma Função 219
Introdução aos Parâmetros 220
Usando Parâmetros Múltiplos 221
Compreendendo as Declarações de Parâmetros em Programas C Mais Antigos 222
Retornando um Valor de uma Função 223
Compreendendo o Comando return 224
Compreendendo os Protótipos de Função 225
Compreendendo a Biblioteca de Execução 226
Compreendendo os Parâmetros Formais e Reais 227
Solucionando os Conflitos nos Nomes 228
Funções Que Não Retornam int 229
Compreendendo as Variáveis Locais 230
Como as Funções Usam a Pilha 231
Compreendendo a Sobrecarga da Função 232
Compreendendo Onde C Coloca as Variáveis Locais 233
Declarando Variáveis Globais 234
Evite Usar Variáveis Globais 235
Solucionando os Conflitos de Nomes de Variáveis Locais e Globais 236
Definindo Melhor o Escopo de uma Variável Global 237
Compreendendo a Chamada por Valor 238
Evitando a Alteração no Valor do Parâmetro com a Chamada por Valor 239
Compreendendo a Chamada por Referência 240

Obtendo um Endereço 241
Usando um Endereço de Memória 242
Alterando o Valor de um Parâmetro 243
Alterando Somente Parâmetros Específicos 244
A Chamada por Referência Ainda Usa a Pilha 245
Introduzindo as Variáveis de Funções que Lembram 246
Compreendendo Como C Inicializa Variáveis Estáticas 247
Usando a Seqüência de Chamada Pascal 248
Compreendendo o Efeito da Palavra-Chave Pascal 249
Escrevendo um Exemplo de Linguagem Mista 250
Compreendendo a Palavra-Chave cdecl 251
Compreendendo a Recursão 252
Compreendendo a Função Recursiva Fatorial 253
Programando Outro Exemplo Recursivo 254
Exibindo Valores para Compreender Melhor a Recursão 255
Compreendendo a Recursão Direta e Indireta 256
Decidindo Usar ou Não a Recursão 257
Compreendendo Por Que as Funções Recursivas São Lentas 258
Compreendendo Como Remover a Recursão 259
Passando Strings para as Funções 260
Passando Elementos Específicos da Matriz 261
Compreendendo const em Parâmetros Formais 262
Usar const Não Impede a Modificação do Parâmetro 263
Compreendendo as Declarações de Strings Não-Limitadas 264
Usando Ponteiros *versus* Declarações de String 265
Como C Usa a Pilha para os Parâmetros String 266
Compreendendo as Variáveis Externas 267
Colocando extern em Uso 268
Compreendendo a Variável Estática Externa 269
Compreendendo a Palavra-chave volatile 270
Compreendendo a Estrutura de Chamada e o Ponteiro de Base 271
Chamando uma Função em Linguagem Assembly 272
Retornando um Valor a partir de uma Função em Linguagem Assembly 273
Introduzindo Funções Que Não Retornam Valores 274
Compreendendo as Funções Que Não Usam Parâmetros 275
Compreendendo a Palavra-chave auto 276
Compreendendo o Escopo 277
Compreendendo as Categorias de Escopo de C 278
Compreendendo o Espaço do Nome e os Identificadores 279
Compreendendo a Visibilidade do Identificador 280
Compreendendo a Duração 281
Funções Que Suportam um Número Variável de Parâmetros 282
Suportando um Número Variável de Parâmetros 283
Como va_start, va_arg e va_end Funcionam 284
Criando Funções Que Suportam Muitos Parâmetros e Tipos 285

OPERAÇÕES DE TECLADO

Lendo um Caractere do Teclado 286
Exibindo um Caractere de Saída 287
Compreendendo a Entrada Bufferizada 288
Atribuindo Entrada do Teclado a uma String 289
Combinando getchar e putchar 290
Lembre-se, getchar e putchar São Macros 291
Lendo um Caractere Usando E/S Direta 292
Entrada Direta no Teclado sem a Exibição do Caractere 293
Sabendo Quando Usar '\r' e '\n' 294
Executando Saída Direta 295
Colando uma Tecla Digitada de Volta no Buffer do Teclado 296
Saída Formatada Rápida Usando cprintf 297
Entrada Rápida Formatada a partir do Teclado 298
Escrevendo uma String de Caracteres 299
Escrita Mais Rápida de String Usando E/S Direta 300
Lendo uma String de Caracteres do Teclado 301
Executando Entrada de String Mais Rápida pelo Teclado 302
Exibindo Saída em Cores 303
Limpando a Tela do Vídeo 304
Apagando até o Final da Linha Atual 305
Apagando a Linha Atual da Tela 306
Posicionando o Cursor para Saída na Tela 307
Determinando a Posição de Linha e de Coluna 308
Inserindo uma Linha em Branco na Tela 309
Copiando Texto da Tela para um Buffer 310
Escrevendo um Buffer de Texto em uma Posição Específica da Tela 311
Determinando as Definições do Modo Texto 312
Controlando as Cores da Tela 313
Atribuindo Cores de Fundo 314
Definindo a Cor de Frente Usando TextColor 315
Definindo a Cor de Fundo Usando textbackground 316
Controlando a Intensidade do Texto 317
Determinando o Modo Atual do Texto 318
Movendo Texto da Tela de um Local para Outro 319
Definindo uma Janela de Texto 320
Usando o Valor Absoluto de uma Expressão Inteira 321
Usando o Arco Co-seno 322
Usando o Arco Seno 323
Usando o Arco Tangente 324
Obtendo o Valor Absoluto de um Número Complexo 325
Arredondando para Cima um Valor em Ponto Flutuante 326
Usando o Co-seno de um Ângulo 327
Usando o Co-seno Hiperbólico de um Ângulo 328
Usando o Seno de um Ângulo 329
Usando o Seno Hiperbólico de um Ângulo 330
Usando a Tangente de um Ângulo 331
Usando a Tangente Hiperbólica de um Ângulo 332
Realizando a Divisão Inteira 333
Trabalhando com Exponencial 334
Usando o Valor Absoluto de uma Expressão em Ponto Flutuante 335
Usando o Resto em Ponto Flutuante 336
Usando a Mantissa e o Expoente de um Valor em Ponto Flutuante 337
Calculando o Resultado de x * 2e 338
Calculando o Logaritmo Natural 339
Calculando o Resultado de log10x 340
Determinando os Valores Máximo e Mínimo 341
Quebrando um Double em Seus Componentes Inteiro e Real 342
Calculando o Resultado de x^n 343
Calculando o Resultado de 10^x 344
Gerando um Número Aleatório 345
Mapeando Valores Aleatórios para um Intervalo Específico 346
Inicializando o Gerador de Números Aleatórios 347
Calculando a Raiz Quadrada de um Valor 348
Criando uma Rotina de Tratamento de Erro Matemático 349

ARQUIVOS, DIRETÓRIOS E DISCOS

Determinando a Unidade de Disco Atual 350
Selecionando a Unidade Atual 351
Determinando o Espaço Disponível no Disco 352
Cuidado com o dblspace 353
Lendo as Informações de uma Tabela de Alocação de Arquivos 354
Compreendendo a Identificação do Disco 355
Efetuando uma Leitura ou Gravação Absoluta de Setor 356
Efetuando E/S no Disco Baseada no BIOS 357
Testando a Prontidão de uma Unidade de Disquete 358
Abrindo um Arquivo Usando fopen 359
Compreendendo a Estrutura FILE 360
Fechando um Arquivo Aberto 361
Lendo e Gravando Informações no Arquivo um Caractere de Cada Vez 362
Compreendendo o Ponteiro de Posição do Ponteiro de Arquivo 363
Determinando a *Posição Atual no Arquivo* 364
Compreendendo os Canais de Arquivos 365
Compreendendo as Traduções de Arquivos 366
Compreendendo a Linha FILES=valor no Config.Sys 367
Usando E/S em Arquivos de Baixo Nível e de Alto Nível 368

Compreendendo os Indicativos de Arquivos 369
Compreendendo a Tabela de Arquivos do Processo 370
Visualizando as Entradas da Tabela de Arquivos do Processo ... 371
Compreendendo a Tabela de Arquivos do Sistema 372
Exibindo a Tabela do Sistema de Arquivos 373
Derivando Indicativos de Arquivos a partir de Ponteiros Stream (de Canais) 374
Executando Saída Formatada em Arquivo 375
Renomeando um Arquivo 376
Excluindo um Arquivo 377
Determinando Como um Programa Pode Acessar um Arquivo 378
Definindo o Modo de Acesso de um Arquivo 379
Ganhando Melhor Controle dos Atributos do Arquivo 380
Testando o Erro no Canal de um Arquivo 381
Determinando o Tamanho de um Arquivo 382
Esvaziando um Canal de E/S 383
Fechando Todos os Arquivos Abertos de uma Só Vez 384
Obtendo o Indicativo de Arquivo de um Canal de Arquivo 385
Criando um Nome de Arquivo Temporário Usando P_tmpdir .. 386
Criando um Nome de Arquivo Temporário Usando TMP ou TEMP 387
Criando um Arquivo Verdadeiramente Temporário 388
Removendo Arquivos Temporários 389
Pesquisando o Caminho de Comandos para um Arquivo 390
Pesquisando um Arquivo na Lista de Subdiretório de um Item do Ambiente 391
Abrindo Arquivos no Diretório TEMP 392
Minimizando as Operações de E/S em Arquivo 393
Escrevendo Código Que Usa Barras Invertidas nos Nomes de Diretório 394
Modificando o Diretório Atual 395
Criando um Diretório 396
Removendo um Diretório 397
Removendo uma Árvore de Diretório 398
Construindo o Nome de Caminho Completo 399
Subdividindo um Caminho de Diretório 400
Construindo um Nome de Caminho 401
Abrindo e Fechando um Arquivo Usando Funções de Baixo Nível 402
Criando um Arquivo 403
Efetuando Operações de Leitura e Gravação de Baixo Nível 404
Testando o Final de um Arquivo 405
Colocando as Rotinas de Arquivo de Baixo Nível para Trabalhar 406
Especificando o Modo para uma Tradução Arquivo-Indicativo .. 407
Posicionando o Ponteiro de Arquivo Usando lseek 408
Abrindo Mais de 20 Arquivos 409
Usando serviços de arquivo baseados no DOS. 410
Obtendo a Data e a Hora de Criação de um Arquivo 411
Obtendo a Data e a Hora de um Arquivo Usando Campos de Bit 412
Definindo a Data e a Hora de um Arquivo 413
Definindo a Data e a Hora de um Arquivo com a Data e a Hora Atuais 414
Lendo e Gravando Dados uma Palavra de Cada Vez 415
Alterando o Tamanho de um Arquivo 416
Controlando as Operações de Leitura e Gravação em Arquivos Abertos 417
Atribuindo um Buffer de Arquivo 418
Alocando um Buffer de Arquivo 419
Criando um Nome de Arquivo Exclusivo Usando mktemp 420
Lendo e Gravando Estruturas 421
Lendo Dados de uma Estrutura a partir de um Canal de Arquivo 422
Duplicando o Indicativo de um Arquivo 423
Forçando a Definição de um Indicativo de Arquivo 424
Associando o Indicativo de um Arquivo com um Canal 425
Compreendendo o Compartilhamento de Arquivo 426
Abrindo um Arquivo para o Acesso Compartilhado 427
Bloqueando o Conteúdo de um Arquivo 428
Ganhando Controle Mais Refinado do Bloqueio de Arquivo 429
Trabalhando com Diretórios do DOS 430
Abrindo um Diretório 431
Lendo uma Entrada do Diretório 432
Usando Serviços de Diretório para Ler C:\Windows 433
Voltando para o Início de um Diretório 434
Lendo Recursivamente os Arquivos de um Disco 435
Determinando a Posição Atual no Arquivo 436
Abrindo um Canal de Arquivo Compartilhado 437
Criando um Arquivo Exclusivo em um Diretório Específico 438
Criando um Novo Arquivo 439
Usando os Serviços do DOS para Acessar um Arquivo 440
Forçando a Abertura de um Arquivo no Modo Binário ou Texto 441
Lendo Linhas de Texto 442
Gravando Linhas de Texto 443
Colocando fgets e fputs em Uso 444
Forçando a Tradução de Arquivo Binário 445
Compreendendo Por Que o Programa copiatxt Não Pode Copiar Arquivos Binários 446
Testando o Final do Arquivo 447
Devolvendo um Caractere 448
Lendo Dados Formatados de Arquivo 449
Posicionamento do Ponteiro de Arquivo com Base em Sua Posição Atual 450
Obtendo Informações do Indicativo de Arquivo 451
Reabrindo um Canal de Arquivo 452

MATRIZES, PONTEIROS E ESTRUTURAS

Compreendendo as Matrizes 453
Declarando uma Matriz 454
Visualizando uma Matriz 455
Compreendendo os Requisitos de Armazenamento de uma Matriz 456
Inicializando uma Matriz 457
Acessando Elementos da Matriz 458
Percorrendo em um Laço os Elementos da Matriz 459
Usando Constantes para Definir as Matrizes 460
Passando uma Matriz a uma Função 461
Revisitando as Matrizes Como Funções 462
Compreendendo Como as Matrizes de String Diferem 463
Passando Matrizes para a Pilha 464
Determinando Quantos Elementos uma Matriz Pode Armazenar 465
Usando o Modelo de Memória Huge para as Matrizes Grandes 466
A Barganha entre Matrizes e Memória Dinâmica 467
Compreendendo as Matrizes Multidimensionais 468
Compreendendo as Linhas e Colunas 469
Acessando Elementos em uma Matriz Bidimensional 470
Inicializando Elementos em uma Matriz Bidimensional 471
Determinando o Consumo de Memória de uma Matriz Multidimensional 472
Percorrendo em um Laço uma Matriz Bidimensional 473
Percorrendo uma Matriz Tridimensional 474
Inicializando Matrizes Multidimensionais 475
Passando uma Matriz Bidimensional para uma Função 476
Tratando as Matrizes Multidimensionais Como uma Dimensão . 477
Compreendendo Como C Armazena Matrizes Multidimensionais 478
Compreendendo a Ordem Linha por Linha versus Coluna por Coluna 479
Matrizes de Estruturas de Matrizes 480
Compreendendo as Uniões 481
Economizando Memória com as Uniões 482
Usando REGS - Uma União Clássica 483
Colocando a União *REGS* em Uso 484
Compreendendo as Estruturas de Campos de Bit 485
Visualizando uma Estrutura de Campos de Bit 486

Compreendendo o Intervalo de Valores de uma Estrutura Bit a Bit 487
Procurando um Valor Específico em uma Matriz 488
Compreendendo uma Pesquisa Binária 489
Usando uma Pesquisa Binária 490
Classificando uma Matriz 491
Compreendendo o Método da Bolha 492
Colocando o Algoritmo da Bolha em Uso 493
Compreendendo o Algoritmo da Seleção 494
Colocando em Uso o Método da Seleção 495
Compreendendo o Algoritmo Shell 496
Colocando em Uso o Algoritmo Shell 497
Compreendendo o Quick Sort 498
Colocando o Quick Sort em Uso 499
Problemas com as Soluções de Classificação Anteriores 500
Classificando uma Matriz de Strings de Caracteres 501
Pesquisando uma Matriz com lfind 502
Procurando Valores com lsearch 503
Pesquisando uma Matriz Classificada com bsearch 504
Classificando Matrizes com qsort 505
Determinando o Número de Elementos na Matriz 506
Compreendendo Ponteiros como Endereços 507
Determinando o Endereço de uma Variável 508
Compreendendo como C Trata Matrizes como Ponteiros 509
Aplicando o Operador de Endereço (&) a uma Matriz 510
Declarando Variáveis Ponteiro 511
Desreferenciando um Ponteiro 512
Usando Valores de Ponteiro 513
Usando Ponteiros com Parâmetros de Função 514
Compreendendo a Aritmética de Ponteiros 515
Incrementando e Decrementando um Ponteiro 516
Combinando uma Referência e Incremento de Ponteiro 517
Percorrendo uma String Usando um Ponteiro 518
Usando Funções Que Retornam Ponteiros 519
Criando uma Função Que Retorna um Ponteiro 520
Compreendendo uma Matriz de Ponteiros 521
Visualizando uma Matriz de Strings de Caracteres 522
Percorrendo em um Laço uma Matriz de Strings de Caracteres 523
Tratando uma Matriz de String de Caracteres Como um Ponteiro 524
Usando um Ponteiro para um Ponteiro para Strings de Caracteres 525
Declarando uma Constante String Usando um Ponteiro 526
Compreendendo o Ponteiro do Tipo void 527
Criando Ponteiros para as Funções 528
Usando um Ponteiro para uma Função 529
Usando um Ponteiro para um Ponteiro para um Ponteiro 530
Compreendendo as Estruturas 531
Uma Estrutura É um Gabarito para a Declaração de Variáveis 532
O Descritor de uma Estrutura É o Nome da Estrutura 533
Declarando uma Variável Estrutura de Modos Diferentes 534
Compreendendo os Membros da Estrutura 535
Visualizando uma Estrutura 536
Pondo uma Estrutura em Uso 537
Passando uma Estrutura a uma Função 538
Alterando uma Estrutura Dentro de uma Função 539
Compreendendo a Indireção (*ponteiro).membro 540
Usando o Formato ponteiro->membro 541
Usando uma Estrutura sem Nome 542
Compreendendo o Escopo de Definição de uma Estrutura 543
Inicializando uma Estrutura 544
Efetuando E/S em Estruturas 545
Usando uma Estrutura Dentro de Outra 546
Estruturas Que Contêm Matrizes 547
Criando uma Matriz de Estruturas 548

SERVIÇOS DOS E BIOS

Compreendendo os Serviços do Sistema do DOS 549
Compreendendo os Serviços da BIOS 550
Compreendendo os Registradores 551
Compreendendo o Registrador de Flags 552
Compreendendo as Interrupções de Software 553
Usando a BIOS para Acessar a Impressora 554
Informação de Ctrl+Break 555
Compreendendo Possíveis Efeitos Colaterais do DOS 556
Suspendendo um Programa Temporariamente 557
Divertindo-se com Som 558
Obtendo Informações Específicas do País 559
Compreendendo o Endereço de Transferência do Disco 560
Acessando e Controlando a Área de Transferência do Disco 561
Usando os Serviços de Teclado da BIOS 562
Obtendo a Lista de Equipamentos da BIOS 563
Controlando a E/S da Porta Serial 564
Acessando os Serviços do DOS Usando bdos 565
Obtendo Informações Estendidas de Erro do DOS 566
Determinando a Quantidade de Memória Convencional da BIOS 567
Construindo um Ponteiro far 568
Dividindo um Endereço far em um Segmento e Deslocamento 569
Determinando a Memória do Núcleo Livre 570
Lendo as Configurações do Registrador de Segmento 571

GERENCIAMENTO DE MEMÓRIA

Compreendendo os Tipos de Memória 572
Compreendendo a Memória Convencional 573
Compreendendo o Layout da Memória Convencional 574
Acessando a Memória Convencional 575
Compreendendo Por Que o PC e o DOS Estão Restritos a 1Mb 576
Produzindo um Endereço a partir de Segmentos e Deslocamentos 577
Compreendendo a Memória Expandida 578
Usando a Memória Expandida 579
Compreendendo a Memória Estendida 580
Compreendendo os Modos Real e Protegido 581
Acessando a Memória Estendida 582
Compreendendo a Área de Memória Alta 583
Compreendendo a Pilha 584
Compreendendo Diferentes Configurações da Pilha 585
Determinando o Tamanho da Pilha Atual do Seu Programa 586
Controlando o Espaço na Pilha com _stklen 587
Atribuindo um Valor a um Intervalo de Memória 588
Copiando um Intervalo de Memória para Outro 589
Copiando um Intervalo de Memória até um Byte Específico 590
Comparando Duas Matrizes de unsigned char 591
Permutando Bytes de Strings de Caracteres Adjacentes 592
Alocando Memória Dinâmica 593
Revisitando a Conversão 594
Liberando a Memória Quando Ela Não É Mais Necessária 595
Alocando Memória Usando a Função calloc 596
Compreendendo o Heap 597
Contornando o Limite de 64Kb para o Heap 598
Alocando Memória a partir da Pilha 599
Alocando Dados Huge 600
Alterando o Tamanho de um Bloco Alocado 601
Compreendendo brk 602
Validando o Heap 603
Efetuando uma Verificação Rápida do Heap 604
Preenchendo o Espaço Livre do Heap 605
Verificando uma Entrada Específica no Heap 606
Percorrendo as Entradas do Heap 607
Examinando o Conteúdo de uma Posição de Memória Específica 608
Colocando Valores na Memória 609
Compreendendo as Portas do PC 610
Acessando os Valores das Portas 611
Compreendendo a CMOS 612
Compreendendo os Modelos de Memória 613
Compreendendo o Modelo de Memória Tiny 614

Compreendendo o Modelo de Memória Small 615
Compreendendo o Modelo de Memória Medium 616
Compreendendo o Modelo de Memória Compact 617
Compreendendo o Modelo de Memória Large 618
Compreendendo o Modelo de Memória Huge 619
Determinando o Modelo de Memória Atual 620

DATA E HORA

Obtendo a Data e a Hora Atuais Como Segundos desde 1/1/1970 .. 621
Convertendo uma Data e uma Hora de Segundos para ASCII 622
Ajuste do Horário de Verão 623
Retardando durante um Determinado Número de Milissegundos 624
Determinando o Tempo de Processamento do Seu Programa ... 625
Comparando Dois Horários 626
Obtendo uma String de Data 627
Obtendo uma String com o Horário 628
Lendo o Temporizador da BIOS 629
Trabalhando com o Horário Local 630
Trabalhando com o Horário de Greenwich 631
Obtendo o Horário do Sistema DOS 632
Obtendo a Data do Sistema do DOS 633
Definindo o Horário do Sistema DOS 634
Definindo a Data do Sistema do DOS 635
Convertendo uma Data do DOS para o Formato do UNIX 636
Usando timezone para Calcular a Diferença entre a Zona Horária 637
Determinando a Zona Horária Atual 638
Definindo Campos de Zona Horária com tzset 639
Usando a Variável de Ambiente TZ 640
Definindo a Entrada de Ambiente TZ a partir de Dentro de Seu Programa 641
Obtendo as Informações de Zona Horária 642
Definindo a Hora do Sistema em Segundos desde a Meia-noite de 1/1/1970 643
Convertendo uma Data para Segundos desde a Meia-noite de 1/1/1970 644
Determinando a Data Juliana de uma Data 645
Criando uma String Formatada de Data e Hora 646
Compreendendo os Tipos de Relógio do PC 647

REDIRECIONANDO A E/S E PROCESSANDO LINHAS DE COMANDO

Aguardando a Digitação de uma Tecla 648
Pedindo uma Senha ao Usuário 649
Escrevendo Sua Própria Função de Senha 650
Compreendendo o Redirecionamento da Saída 651
Compreendendo o Redirecionamento da Entrada 652
Combinando o Redirecionamento da Entrada e da Saída 653
Usando stdout e stdin 654
Compreendendo o Operador de Canalização 655
Compreendendo getchar e putchar 656
Numerando a Entrada Redirecionada 657
Garantindo Que uma Mensagem Apareça na Tela 658
Escrevendo Seu Próprio Comando more 659
Exibindo um Contador de Linhas Redirecionadas 660
Exibindo um Contador de Caracteres Redirecionados 661
Criando um Comando More com Controle de Tempo 662
Impedindo o Redirecionamento da E/S 663
Usando o Indicativo de Arquivo stdprn 664
Desviando a Saída Redirecionada para um Arquivo 665
Usando o Indicativo de Arquivo stdaux 666
Encontrando Ocorrências de Substring Dentro da Entrada Redirecionada 667
Exibindo as Primeiras n Linhas da Entrada Redirecionada 668
Compreendendo os Argumentos da Linha de Comando 669
Exibindo um Contador de Argumentos da Linha de Comandos . 670
Exibindo a Linha de Comando 671
Trabalhando com Argumentos da Linha de Comando entre Aspas 672
Exibindo o Conteúdo de um Arquivo a partir da Linha de Comando 673
Tratando argv Como um Ponteiro 674
Compreendendo Como C Sabe sobre a Linha de Comando 675
Compreendendo o Ambiente 676
Tratando env Como um Ponteiro 677
Use void para os Parâmetros de main 678
Trabalhando com Números da Linha de Comando 679
Compreendendo os Valores de Status da Saída 680
Usando return para o Processamento do Status da Saída 681
Determinando Se main Deve ou Não Ser Declarada Como void 682
Pesquisando o Item Específico no Ambiente 683
Como o DOS Trata o Ambiente 684
Usando a Variável Global environ 685
Acrescentando um Item ao Ambiente Atual 686
Acrescentando Elementos no Ambiente DOS 687
Abortando o Programa Atual 688
Definindo Funções Que Executam ao Término do Programa ... 689
Compreendendo as Bibliotecas 690
Reutilizando o Código-Objeto 691
Problemas com a Compilação de Arquivos C e OBJ 692
Criando um Arquivo de Biblioteca 693
Compreendendo as Operações da Biblioteca Comum 694
Listando as Rotinas em um Arquivo de Biblioteca 695
Use Bibliotecas para Reduzir Seu Tempo de Compilação 696
Aprendendo Mais sobre as Capacidades da Sua Biblioteca 697
Compreendendo o Linkeditor 698
Vendo as Capacidades do Linkeditor 699
Usando um Mapa do Linkeditor 700
Usando Arquivos de Resposta do Linkeditor 701
Simplificando a Criação de Aplicativos com o MAKE 702
Criando um Arquivo Simples do MAKE 703
Usando Múltiplos Arquivos de Dependência com o MAKE 704
Comentando Seus Arquivos do MAKE 705
Linhas de Comando e MAKE 706
Colocando Múltiplas Dependências em um Arquivo do MAKE 707
Regras Implícitas e Explícitas do MAKE 708
Usando Macros do MAKE 709
Macros Predefinidas do MAKE 710
Executando Processamento Condicional com o MAKE 711
Testando um Nome de Macro 712
Incluindo um Segundo Arquivo do MAKE 713
Usando os Modificadores de Macro do MAKE 714
Finalizando um Arquivo do MAKE com um Erro 715
Desabilitando a Exibição do Nome do Comando 716
Usando o Arquivo BUILTINS.MAK 717
Executando Processamento de Status de Saída no MAKE 718
Chamando e Alterando uma Macro ao Mesmo Tempo 719
Executando um Comando MAKE para Múltiplos Arquivos Dependentes 720

C AVANÇADO

Determinando Se o Co-Processador Matemático Está Presente .. 721
Compreendendo as Macros ctype.h e istype 722
Controlando o Vídeo Direto 723
Detectando Erros Matemáticos e do Sistema 724
Exibindo Mensagens de Erro Predefinidas 725
Determinando o Número de Versão do Sistema Operacional ... 726
Compreendendo a Portabilidade 727
Efetuando um Goto Não-local 728
Obtendo a Identificação do Processo 729
Chamando um Comando Interno do DOS 730
Usando a Variável Global _psp 731
Usando o Modificador const nas Declarações de Variáveis 732
Usando Tipos Enumerados 733
Colocando um Tipo Enumerado em Uso 734

Compreendendo um Valor Enumerado 735
Atribuindo um Valor Específico a um Tipo Enumerado 736
Salvando e Restaurando os Registradores 737
Introdução às Listas Dinâmicas 738
Declarando uma Estrutura de Lista Ligada 739
Criando uma Lista Ligada 740
Exemplo de uma Lista Ligada Simples 741
Compreendendo como Percorrer uma Lista Ligada 742
Criando uma Lista Mais Útil 743
Anexando um Elemento na Lista 744
Inserindo um Elemento na Lista 745
Exibindo um Diretório Classificado 746
Excluindo um Elemento de uma Lista 747
Usando uma Lista Duplamente Ligada 748
Criando uma Lista Duplamente Ligada Simples 749
Compreendendo nodo->anterior->próximo 750
Removendo um Elemento de uma Lista Duplamente Ligada ... 751
Inserindo um Elemento em uma Lista Duplamente Ligada 752
Compreendendo os Processos-Filho 753
Gerando um Processo-Filho 754
Usando Outras Funções spawnlxx 755
Usando as Funções spawnvxx 756
Executando um Processo-Filho 757
Usando Outras Funções execlxx 758
Usando as Funções execvxx 759
Compreendendo os Overlays 760
Compreendendo as Interrupções 761
As Interrupções do PC 762
Usando a Palavra-Chave interrupt 763
Determinando o Vetor de uma Interrupção 764
Definindo um Vetor de Interrupção 765
Habilitando e Desabilitando as Interrupções 766
Criando uma Rotina Simples de Tratamento de Interrupção 767
Encadeando uma Segunda Interrupção 768
Gerando uma Interrupção 769
Interceptando o Temporizador do PC 770
Compreendendo os Erros Críticos 771
Tratamento de Erro Crítico em C 772
Uma Rotina de Tratamento de Erro Crítico Completa 773
Restaurando as Interrupções Alteradas 774
Criando uma Rotina de Tratamento de Ctrl+Break 775
Usando os Serviços do DOS na Sua Rotina de Tratamento de Erro Crítico 776
Aumentando o Desempenho com o Uso da Seleção do Conjunto de Instruções 777
Funções Intrínsecas In-line 778
Habilitando e Desabilitando as Funções Intrínsecas 779
Compreendendo as Chamadas Rápidas de Funções 780
Regras para a Passagem de Parâmetros _fastcall 781
Compreendendo o Código Invariante 782
Compreendendo a Supressão de Carga Redundante 783
Compreendendo a Compactação do Código 784
Compreendendo a Compactação do Laço 785
Compreendendo a Indução do Laço e a Redução da Força 786
Compreendendo a Eliminação de Subexpressão Comum 787
Compreendendo as Conversões Comuns de C Padrão 788
Compreendendo os Quatro Tipos Básicos de C 789
Compreendendo os Tipos Fundamentais versus os Derivados ... 790
Compreendendo os Inicializadores 791
Compreendendo o Sistema de Ligação (Linkage) 792
Compreendendo as Declarações Experimentais 793
Contrastando Declarações e Definições 794
Compreendendo lvalues 795
Compreendendo rvalues 796
Usando Palavras-chave de Registradores de Segmento 797
Use os Ponteiros far com Cuidado 798
Compreendendo os Ponteiros Normalizados 799
Comandos do Co-Processador Matemático 800
Compreendendo cdecl e pascal nas Variáveis 801
Impedindo as Inclusões Circulares 802

INTRODUÇÃO À C++

Introduzindo C++ 803
Como os Arquivos-Fonte de C++ São Diferentes 804
Iniciando com um Programa Simples em C++ 805
Compreendendo o Canal de Entrada/Saída cout 806
Escrevendo Valores e Variáveis com cout 807
Combinando Diferentes Tipos de Valores com cout 808
Exibindo Valores Hexadecimais e Octais 809
Redirecionando cout 810
Se Você Gosta de printf, Use printf 811
Escrevendo Saída em cerr 812
Recebendo Entrada com cin 813
cin Não Usa Ponteiros 814
Compreendendo Como cin Seleciona os Campos de Dados 815
Compreendendo Como os Canais de E/S Conhecem os Tipos dos Valores 816
Efetuando Saída Usando clog 817
cin, cout, cerr e clog São Ocorrências de Classe 818
Descarregando a Saída com flush 819
Compreendendo o Que iostream.h Contém 820
C++ Requer Protótipos de Funções 821
C++ Acrescenta Novas Palavras-Chave 822
C++ Suporta as Uniões Anônimas 823
Resolvendo o Escopo Global 824
Fornecendo Valores Padrão de Parâmetros 825
Controlando a Largura da Saída de cout 826
Usando setw para Definir a Largura para cout 827
Especificando um Caractere de Preenchimento em cout 828
Justificando à Direita e à Esquerda a Saída de cout 829
Controlando o Número de Dígitos em Ponto Flutuante Exibidos por cout 830
Exibindo Valores nos Formatos Fixo ou Científico 831
Restaurando cout para o Padrão 832
Definindo a Base da E/S 833
Declarando Variáveis Onde Você Precisa Delas 834
Colocando Valores de Parâmetro Padrão em Protótipos de Função 835
Usando Operadores Bit a Bit e cout 836
Compreendendo a Avaliação Preguiçosa (ou Curto- Circuito) ... 837
Usando a Palavra-Chave const em C++ 838
Usando a Palavra-Chave enum em C++ 839
Compreendendo o Espaço Livre 840
Alocando Memória com new 841
Alocando Múltiplas Matrizes 842
Testando a Falta de Espaço Livre 843
Considerações sobre o Espaço no Heap 844
Usando Ponteiros far e o Operador new 845
Liberando Memória de Volta ao Espaço Livre 846
Compreendendo as Referências de C++ 847
Passando uma Referência a uma Função 848
Atentando para os Objetos Ocultos 849
Usando Três Modos de Passar Parâmetros 850
Regras para Trabalhar com Referências 851
As Funções Podem Retornar Referências 852
Usando a Palavra-Chave inline de C++ 853
Usando a Palavra-chave asm de C++ 854
Lendo um Caractere Usando cin 855
Escrevendo um Caractere com cout 856
Escrevendo um Programa Simples de Filtro 857
Escrevendo um Comando Tee Simples 858
Escrevendo um Comando First Simples 859
Escrevendo um Comando First Melhor 860
Testando o Final do Arquivo 861
Gerando uma Nova Linha com endl 862
Compreendendo as Especificações de Ligação 863
Compreendendo a Sobrecarga 864
Sobrecarregando as Funções 865
Sobrecarregando Funções: um Segundo Exemplo 866
Evitando a Ambigüidade da Sobrecarga 867
Lendo uma Linha de Cada Vez com cin 868

Usando cin.getline em um Laço 869
Alterando o Tratamento Normal do Operador new 870
Definindo uma Nova Rotina de Tratamento com set_new_handler 871
Determinando uma Compilação C++ 872
Compreendendo as Estruturas em C++ 873
Introduzindo Funções Como Membros da Estrutura 874
Definindo uma Função-membro Dentro de uma Estrutura 875
Declarando uma Função-membro Fora de uma Estrutura . . . 876
Passando Parâmetros para uma Função-membro 877
Múltiplas Variáveis da Mesma Estrutura 878
Diferentes Estruturas com os Mesmos Nomes de Função-membro 879
Diferentes Funções com os Mesmos Nomes de Membro . . . .

Classes

Compreendendo os Objetos 881
Compreendendo a Programação Orientada a Objetos . . . 882
Compreendendo Por Que Você Deve Usar Objetos 883
Quebrando os Programas em Objetos 884
Compreendendo Objetos e Classes . 885
Compreendendo as Classes de C++ 886
Compreendendo o Encapsulamento 887
Compreendendo o Polimorfismo 888
Compreendendo a Herança 889
Decidindo entre Classes e Estruturas 890
Criando um Modelo Simples de Classe 891
Implementando o Programa da Classe Simples 892
Definindo os Componentes de uma Classe 893
Compreendendo o Operador de Definição de Escopo . 894
Usando ou Omitindo o Nome da Classe das Declarações 895
Compreendendo o Rótulo public: 896
Compreendendo a Ocultação de Informações 897
Compreendendo o Rótulo private: 898
Usando o Rótulo protected: 899
Usando Dados Públicos e Privados 900
Determinando o Que Ocultar e o Que Tornar Público 901
Os Métodos Públicos São Frequentemente Chamados de Funções de Interface 902
Definindo Funções de Classe Fora da Classe 903
Definindo Métodos Dentro e Fora das Classes 904
Compreendendo as Ocorrências do Objeto 905
As Ocorrências de Objetos Devem Compartilhar o Código 906
Acessando os Membros da Classe 907
Revisando o Operador de Definição Global 908
Inicializando os Valores da Classe 909
Usando Outro Método para Inicializar os Valores de Classe . . 910
Compreendendo os Membros de Classe Estática 911
Usando Dados-Membro Estáticos 912
Usando Funções-Membro Estáticas 913
Compreendendo as Declarações das Funções-Membro 914
Usando Declarações de Função In-line 915
Determinando Quando Usar Funções In-line ou Não 916
Compreendendo as Classes e as Uniões 917
Introduzindo as Uniões Anônimas 918
Introduzindo as Funções Amigas 919
Introduzindo as Classes Amigas 920

Funções de Classes Comuns

Compreendendo as Funções Construtoras 921
Usando Funções Construtoras com Parâmetros 922
Usando uma Função Construtora 923
Compreendendo Quando um Programa Executa uma Construtora 924
Usando Funções Construtoras com Parâmetros 925
Resolvendo os Conflitos de Nomes em uma Função Construtora 926
Usando uma Construtora para Alocar Memória 927
Tratando a Alocação de Memória de um Modo Mais Claro . 928
Valores de Parâmetro Padrão para as Construtoras 929
Sobrecarregando as Funções Construtoras 930
Encontrando o Endereço de uma Função Sobrecarregada . . 931
Usando Funções Construtoras com um Único Parâmetro 932
Compreendendo as Funções Destrutoras 933
Usando uma Função Destrutora 934
Compreendendo Por Que Você Deve Usar as Funções Destrutoras 935
Compreendendo Quando um Programa Chama uma Função Destrutora 936
Usando uma Construtora de Cópia 937
Usando Construtoras Explícitas 938
Compreendendo o Escopo da Classe 939
Compreendendo as Classes Inseridas Dentro de Outras 940
Compreendendo as Classes Locais 941
Resolvendo os Conflitos de Nome de Membro e de Parâmetro . . 942
Criando uma Matriz de Variáveis de Classe 943
Construtoras e Matrizes de Classes 944
Sobrecarregando um Operador 945
Criando uma Função de Operador de Membro 946
Sobrecarregando o Operador de Adição 947
Sobrecarregando o Operador do Sinal de Subtração 948
Sobrecarregando os Operadores de Incremento de Prefixo e de Sufixo 949
Sobrecarregando os Operadores de Decremento de Prefixo e de Sufixo 950
Revisitando as Restrições na Sobrecarga de Operador 951
Usando uma Função amiga para Sobrecarregar os Operadores . . . 952
Restrições na Sobrecarga de Operador de Função Amiga . . . 953
Usando uma Função amiga para Sobrecarregar os Operadores ++ ou — 954
Razões para Sobrecarregar os Operadores com as Funções amigas 955
Sobrecarregando o Operador new 956
Sobrecarregando o Operador delete 957
Sobrecarregando new e delete para as Matrizes 958
Sobrecarregando o Operador de Matriz [] 959
Sobrecarregando o Operador de Chamada de Função () 960
Sobrecarregando o Operador de Ponteiro –> 961
Sobrecarregando o Operador Vírgula , ... 962
Compreendendo a Abstração 963
Alocando um Ponteiro para uma Classe 964
Descartando um Ponteiro para uma Classe 965
Descartando o Espaço em Branco Preliminar na Entrada 966
Compreendendo as Bibliotecas de Classes 967
Coloque Suas Definições de Classe em um Arquivo de Cabeçalho 968
Usando a Palavra-Chave inline com Funções-membro da Classe 969
Inicializando uma Matriz de Classe 970
Destruindo a Matriz de Classe 971
Criando Matrizes de Classe Inicializadas 972
Inicializando uma Matriz com uma Construtora de Múltiplos Argumentos 973
Criando Matrizes Inicializadas versus Não-inicializadas . . . 974
Trabalhando com Matrizes de Classe 975
Compreendendo Como as Matrizes de Classe Usam a Memória 976
O Código de Classe in-line Permite Modificações 977
Compreendendo o Armazenamento Estático 978

E/S com C++

Sincronizando as Operações de Canal de E/S com stdio . . . 979
Compreendendo os Canais de E/S de C++ 980
Compreendendo os Canais de Saída de C++ 981
Compreendendo os Canais de Entrada de C++ 982
Usando os Membros ios para Formatar a Entrada e a Saída . . . 983
Definindo os Sinalizadores de Formato 984
Limpando os Sinalizadores de Formato 985
Usando a Função setf Sobrecarregada 986
Examinando os Sinalizadores de Formatação Atuais 987

Definindo Todos os Sinalizadores ... 988
Usando a Função precision ... 989
Usando a Função fill ... 990
Compreendendo os Manipuladores ... 991
Usando Manipuladores para Formatar a E/S ... 992
Comparando os Manipuladores e as Funções-Membro ... 993
Criando Suas Próprias Funções Inserçoras ... 994
Sobrecarregando o Operador de Extração ... 995
Outro Modo de Sobrecarregar o Operador de Inserção de cout ... 996
Criando Suas Próprias Funções Extratoras ... 997
Criando um Exemplo de Extratora ... 998
Criando Sua Própria Função Manipuladora ... 999
Criando Manipuladores sem Parâmetros ... 1000
Usando Parâmetros com Manipuladores ... 1001
Compreendendo a Antiga Biblioteca de Classes de Canais ... 1002
Abrindo um Canal de Arquivo ... 1003
Fechando um Canal de Arquivo ... 1004
Lendo e Gravando Dados do Canal de Arquivo ... 1005
Verificando o Status de uma Operação de Arquivo ... 1006
Juntando as Operações de Canal de Arquivo ... 1007
Efetuando uma Operação de Cópia Binária ... 1008
Compreendendo a Classe streambuf ... 1009
Escrevendo um Exemplo Simples com streambuf ... 1010
Lendo Dados Binários Usando read ... 1011
Escrevendo Dados Binários Usando write ... 1012
Usando a Função-membro gcount ... 1013
Usando as Funções get Sobrecarregadas ... 1014
Usando o Método getline ... 1015
Detectando o Final do Arquivo ... 1016
Usando a Função ignore ... 1017
Usando a Função peek ... 1018
Usando a Função putback ... 1019
Encontrando a Posição Atual no Canal de Arquivo ... 1020
Controlando o Ponteiro de Canal de Arquivo ... 1021
Usando seekg e seekp para Acesso Aleatório ... 1022
Manipulando a Posição do Ponteiro de Arquivo Dentro de um Arquivo ... 1023
Determinando o Status Atual de um Canal de E/S ... 1024
Compreendendo as Classes de E/S de Matriz ... 1025
Compreendendo os Canais de String de Caracteres ... 1026
Usando istrstream para Escrever uma String de Caracteres ... 1027
Compreendendo Melhor ostrstream ... 1028
Usando as Formas istrstream Sobrecarregadas ... 1029
Usando pcount com Matrizes de Saída ... 1030
Manipulando Matrizes de Canais com as Funções-Membro ios ... 1031
Usando strstream ... 1032
Efetuando Acesso Aleatório Dentro de uma Matriz stream ... 1033
Usando Manipuladores com Matrizes stream ... 1034
Usando um Operador de Inserção Personalizado com as Matrizes String ... 1035
Usando Operadores de Extração Personalizados com as Matrizes stream ... 1036
Usando Matrizes Dinâmicas com Canais de E/S ... 1037
Compreendendo os Usos para a Formatação da Matriz stream ... 1038
Compreendendo o Manipulador ends ... 1039
Chamando um Objeto a partir de Outro ... 1040
Falando ao Compilador sobre uma Classe ... 1041
Revisitando as Amigas ... 1042
Declarando a Classe Leitor Como uma Amiga ... 1043
Outro Exemplo de Classe Amiga ... 1044
Eliminando a Necessidade do Comando class nome_classe ... 1045
Restringindo o Acesso a uma Amiga ... 1046
Conflito de Nomes e Amigas ... 1047

HERANÇA E POLIMORFISMO

Herança em C++ ... 1048
Compreendendo as Classes Base e Derivada ... 1049
Derivando uma Classe ... 1050
Compreendendo as Construtoras Base e Derivada ... 1051
Pondo os Membros Protegidos em Uso ... 1052
Compreendendo Quando Usar os Membros Protegidos ... 1053
Revisando a Herança de Classe-base Pública ou Privada ... 1054
Compreendendo a Herança de Classe-base Protegida ... 1055
Compreendendo a Herança Múltipla ... 1056
Uma Herança Múltipla Simples ... 1057
Compreendendo a Ordem das Construtoras e as Classes-base ... 1058
Declarando uma Classe-base Como Privada ... 1059
Funções Destrutoras e Herança Múltipla ... 1060
Conflitos de Nomes entre as Classes Base e Derivada ... 1061
Resolvendo Conflitos de Nome de Classe e de Base ... 1062
Compreendendo Quando as Classes Herdadas Executam as Construtoras ... 1063
Exemplo de uma Construtora de Classe Herdada ... 1064
Como Passar Parâmetros para as Construtoras de Classe-base ... 1065
Compreendendo as Declarações de Acesso com as Classes Derivadas ... 1066
Usando Declarações de Acesso com as Classes Derivadas ... 1067
Evitando Ambigüidade com as Classes-base Virtuais ... 1068
Compreendendo as Classes-base Virtuais ... 1069
Amigas Mútuas ... 1070
Como uma Classe Derivada Pode Tornar-se uma Classe-base ... 1071
Usando Membros Protegidos nas Classes Derivadas ... 1072
Definindo Dados de Classe Estática ... 1073
Inicializando um Dado-Membro Estático ... 1074
Acesso Direto de um Dado-Membro Estático ... 1075
Compreendendo os Dados-Membro Privados Estáticos ... 1076
Compreendendo as Funções-Membro Estáticas ... 1077
Acesso Direto de uma Função Estática Pública ... 1078
Usando Tipos Expandidos Como Membros da Classe ... 1079
Embutindo uma Classe Dentro de Outra ... 1080
Compreendendo Subclasses e Superclasses ... 1081
Comandos em Linguagem Assembly In-line em uma Função Método ... 1082
Os Membros de Classe Podem Ser Recursivos ... 1083
Compreendendo o Ponteiro this ... 1084
Como o Ponteiro this Difere dos Outros Ponteiros ... 1085
Compreendendo a Amarração Precoce ou Tardia ... 1086
Ponteiros para Classes ... 1087
Usando o Mesmo Ponteiro para Diferentes Classes ... 1088
Conflitos de Nome Derivado e Base com Ponteiros ... 1089
Compreendendo as Funções Virtuais ... 1090
Herdando o Atributo Virtual ... 1091
As Funções Virtuais São Hierárquicas ... 1092
Implementando o Polimorfismo ... 1093
Compreendendo as Funções Virtuais Puras ... 1094
Compreendendo as Classes Abstratas ... 1095
Usando Funções Virtuais ... 1096
Mais sobre Amarração Precoce ou Tardia ... 1097
Decidindo entre a Amarração Precoce e Tardia ... 1098
Um Exemplo de Amarração Precoce e Amarração Tardia ... 1099
Definindo um Manipulador de Canal de Saída ... 1100
É Hora de Dar uma Olhada em iostream.h ... 1101
Usando sizeof com uma Classe ... 1102
As Palavras-chave Private, Public e Protected Também Podem Ser Aplicadas às Estruturas ... 1103
Compreendendo as Conversões de Classes ... 1104
Convertendo Dados em uma Construtora ... 1105
Atribuindo uma Classe a Outra ... 1106
Use Amigas para a Conversão ... 1107
Determinando Quando os Operadores Aumentam ou Reduzem a Legibilidade ... 1108

FUNÇÕES GENÉRICAS E GABARITOS

Compreendendo os Gabaritos ... 1109
Colocando um Gabarito Simples em Uso ... 1110
Compreendendo Melhor as Funções Genéricas ... 1111
Gabaritos Que Suportam Múltiplos Tipos ... 1112

Mais sobre Gabaritos com Múltiplos Tipos Genéricos 1113
Sobrecarregando Explicitamente uma Função Genérica 1114
Compreendendo as Restrições nas Funções Genéricas 1115
Usando uma Função Genérica 1116
Usando uma Função Genérica do Algoritmo da Bolha 1117
Usando Funções Genéricas para Compactar uma Matriz 1118
Onde Colocar os Gabaritos 1119
Os Gabaritos Também Eliminam as Classes Duplicadas 1120
Compreendendo as Classes Genéricas 1121
Usando Classes Genéricas 1122
Criando uma Classe Genérica com Dois Tipos de Dados Genéricos 1123
Criando um Manipulador Parametrizado 1124
Criando uma Classe de Matriz Genérica 1125

TRATAMENTO DAS EXCEÇÕES E PORTABILIDADE DE TIPO

Compreendendo o Tratamento das Exceções 1126
Compreendendo a Forma de Tratamento Básico da Exceção 1127
Escrevendo uma Rotina de Tratamento de Exceção Simples 1128
Compreendendo o Comando throw 1129
As Exceções São de um Tipo Específico 1130
Lançando Exceções a partir de uma Função Dentro de um Bloco try 1131
Localizando um Bloco try para uma Função 1132
Compreendendo Quando o Programa Executa catch 1133
Usando Múltiplos Comandos catch com um Único Bloco try 1134
Usando o Operador Reticências (...) com as Exceções 1135
Pegando Todas as Exceções Dentro de um Único Bloco try 1136
Pegando Exceções Explícitas e Genéricas em um Único Bloco try 1137
Restringindo as Exceções 1138
Relançando uma Exceção 1139
Aplicando o Tratamento da Exceção 1140
Usando Argumentos Padrão de Função 1141
Evitando Erros com Argumentos de Função Padrão 1142
Argumentos Padrão *versus* Sobrecarga de Função 1143
Criando Funções de Conversão 1144
Usando Funções de Conversão para Aumentar a Portabilidade de Tipo 1145
Funções de Conversão *versus* Operadores Sobrecarregados 1146
Compreendendo os Novos Operadores de Conversão de C++ 1147
Usando o Operador const_cast 1148
Usando o Operador dynamic_cast 1149
Usando o Operador reinterpret_cast 1150
Usando o Operador static_cast 1151
Compreendendo os Namespaces 1152
Usando Namespaces 1153
Usando o Comando using com namespace 1154
Compreendendo a Identificação de Tipo em Tempo de Execução 1155
Usando typeid para a Identificação de Tipo em Tempo de Execução 1156
Compreendendo a Classe type_info 1157
Compreendendo a Palavra-chave mutable 1158
Usando a Palavra-chave mutable Dentro de uma Classe 1159
Considerações sobre a Palavra-chave mutable 1160
Introduzindo o Tipo de Dados bool 1161
Usando o Tipo de Dados bool 1162

CRIANDO EXEMPLOS DE CLASSES REUTILIZÁVEIS

Criando um Tipo String 1163
Definindo as Características do Tipo String 1164
Criando a Classe Strings 1165
Escrevendo as Construtoras para a Classe Strings 1166
Efetuando E/S com a Classe Strings 1167
Escrevendo as Funções de Atribuição para a Classe Strings 1168
Sobrecarregando o Operador + para Concatenar Objetos Strings 1169
Removendo uma String a partir de Dentro de um Objeto Strings 1170
Sobrecarregando os Operadores Relacionais 1171
Determinando o Tamanho de um Objeto Strings 1172
Convertendo um Objeto Strings para uma Matriz de Caracteres 1173
Usando um Objeto Strings Como uma Matriz de Caracteres 1174
Demonstrando o Objeto Strings 1175
Criando um Cabeçalho para a Classe Strings 1176
Outro Exemplo de Strings 1177
Usando uma Classe de C++ para Criar uma Lista Duplamente Ligada 1178
Compreendendo os Membros da Classe dbllinkob 1179
Compreendendo as Funções pegaproximo e pegaanterior 1180
Compreendendo as Funções de Sobrecarga operator 1181
Herdando a Classe Lista_Objeto 1182
Compreendendo a Classe lista_ligada 1183
Compreendendo a Função armazena da lista_ligada 1184
Compreendendo a Função remove de lista_ligada 1185
Compreendendo as Funções pegainicio e pegafim 1186
Exibindo a lista_ligada na Ordem do Começo para o Fim 1187
Exibindo a lista_ligada em Ordem Inversa 1188
Pesquisando a Lista 1189
Implementando um Programa lista_Ligada Simples 1190
Criando uma Classe de Lista Duplamente Ligada Genérica 1191
Compreendendo os Membros Genéricos da Classe lista_objeto 1192
Compreendendo a Classe lista_ligada Genérica 1193
Usando as Classes Genéricas com uma Lista char 1194
Usando as Classes Genéricas com uma Lista Dupla 1195
Usando as Classes Genéricas com uma Estrutura 1196
Sobrecarregando o Operador de Comparação == 1197
Outras Melhorias na Lista Genérica 1198
Usando Objetos com a Função armazena 1199
Escrevendo uma Função para Determinar o Comprimento da Lista 1200

BIBLIOTECA DE GABARITOS PADRÕES

Introduzindo a Biblioteca de Gabaritos Padrões 1201
Compreendendo os Arquivos de Cabeçalho da Biblioteca de Gabaritos Padrões 1202
Compreendendo os Repositórios 1203
Usando um Exemplo de Repositório 1204
Apresentando os Repositórios da Biblioteca de Gabaritos Padrões 1205
Compreendendo os Repositórios Forward e Reversible 1206
Compreendendo os Repositórios de Seqüência da Biblioteca de Gabaritos Padrões 1207
Compreendendo o Comando using namespace std 1208
Compreendendo os Repositórios Associativos da Biblioteca de Gabaritos Padrões 1209
Compreendendo os Iteradores 1210
Usando um Exemplo de Iterador 1211
Compreendendo Melhor os Tipos de Iterador de Entrada e Saída da Biblioteca de Gabaritos Padrões (STL) 1212
Compreendendo os Outros Tipos de Iteradores da Biblioteca de Gabaritos Padrões 1213
Compreendendo os Conceitos 1214
Compreendendo os Modelos 1215
Compreendendo os Algoritmos 1216
Usando Outro Exemplo de Algoritmo da Biblioteca de Gabaritos Padrões 1217
Descrevendo os Algoritmos Que a STL Inclui 1218
Estudando o Algoritmo for_each da STL 1219
Estudando o Algoritmo generate_n da STL 1220
Compreendendo o Algoritmo random_shuffle da STL 1221
Usando o Algoritmo partial_sort_copy 1222
Compreendendo o Algoritmo merge 1223
Compreendendo o Algoritmo inner product 1224
Compreendendo Melhor os Vetores 1225
Usando Outro Programa Simples de Vetor 1226
Comparando os Vetores com as Matrizes de C 1227

Compreendendo o Repositório de Seqüência bit_vector . . . 1228
Usando um Exemplo bvector simples 1229
Compreendendo o Tipo list 1230
Compreendendo os Componentes Genéricos do Repositório list 1231
Construindo um Objeto list 1232
Inserindo Objetos na Lista 1233
Usando a Função-Membro assign 1234
Usando as Funções-Membro remove e empty 1235
Percorrendo o Objeto list 1236
Compreendendo o Tipo slist 1237
Compreendendo as Inserções Dentro de um Repositório de Seqüência slist 1238
Compreendendo o Repositório deque 1239
Usando o Repositório deque 1240
Usando as Funções-membro erase e clear 1241
Usando o Operador de Matriz [] com um deque 1242
Usando os Iteradores *reverse* com um ***deque*** . 1243
Gerenciando o Tamanho do deque 1244
Compreendendo o Objeto map 1245
Um Exemplo Simples de map 1246
Usando Funções Membro para Gerenciar o Map . 1247
Controlando o Tamanho e o Conteúdo do Map 1248
Compreendendo os sets 1249
Um Exemplo Simples de set 1250

INTRODUÇÃO À PROGRAMAÇÃO WINDOWS

Introduzindo a Programação em Win32 1251
Mais Diferenças entre os Programas Windows e DOS 252
Introduzindo os Encadeamentos 253
Compreendendo as Mensagens 1254
Compreendendo os Componentes de uma Janela 1255
Compreendendo as Janelas-mãe e as Janelas-filha . . . 1256
Criando um Programa Windows Genérico 1257
Compreendendo os Arquivos de Recursos 1258
Compreendendo os Indicativos do Windows 1259
Definindo os Tipos de Indicativo do Windows . . 1260
Compreendendo o Arquivo de Cabeçalho Genérico 1261
Compreendendo as Funções Callback 1262
Apresentando a Interface de Programação de Aplicativos do Windows 1263
Examinando Melhor o Programa generico.CPP 1264
Compreendendo a Função WinMain 1265
Compreendendo a Criação de Janela 1266
Compreendendo a Função CreateWindow 1267
Compreendendo a Função ShowWindow 1268
Compreendendo a Função RegisterClass 1269
Aprendendo Mais sobre as Mensagens 1270
Usando TranslateMessage para Processar as Mensagens . . 1271
Usando DispatchMessage para Processar as Mensagens . 1272
Compreendendo os Componentes de um Programa Windows Simples 1273
Compreendendo o Tipo LPCTSTR 1274
Compreendendo o Tipo DWORD . . 1275
Compreendendo as Classes Predefinidas do Windows . 1276
Usando Classes Predefinidas para Criar uma Janela Simples . . . 1277
O Windows Envia WM_CREATE Quando Cria uma Janela 1278
Compreendendo os Estilos Window e Control 1279
Criando Janelas com Estilos Estendidos 1280
Destruindo Janelas 1281
Compreendendo a Função da API RegisterClassEx 1282
Anexando Informações em uma Janela com SetProp 1283
Usando EnumProps para Listar as Propriedades de uma Janela 1284
Compreendendo as Funções de Callback 1285
Compreendendo a Função MessageBox 1286
Compreendendo a Função MessageBeep 1287

MENSAGENS E MENUS

Revisitando as Mensagens 1288
Compreendendo o Fluxo de Mensagens 1289
Compreendendo Melhor os Componentes da Estrutura MSG 1290
Compreendendo a Função PeekMessage 1291
Compreendendo a Função PostMessage 1292
Compreendendo a Função SendMessage 1293
Usando a Função ReplyMessage 1294
Ganchos de Mensagens 1295
Usando a Função SetWindowsHookEx 1296
Compreendendo a Função ExitWindowsEx . . . 1297
Compreendendo os Tipos de Menu 1298
Compreendendo a Estrutura de um Menu 1299
Criando um Menu Dentro de um Arquivo de Recurso 1300
Compreendendo os Descritores POPUP e MENUITEM . 1301
Acrescentando um Menu à Janela de um Aplicativo 1302
Alterando os Menus Dentro de um Aplicativo 1303
Compreendendo as Mensagens Geradas por Menus 1304
Compreendendo a Função LoadMenu 1305
Usando a Função ModifyMenu 1306
Usando EnableMenuItem para Controlar os Menus 1307
Usando AppendMenu para Expandir um Menu 1308
Usando DeleteMenu para Excluir Seleções do Menu 1309
Usando Teclas Aceleradoras com Itens de Menu 1310
Criando uma Tabela Aceleradora de Exemplo 1311
Compreendendo Melhor a Estrutura do Arquivo de Recurso . . . 1312
Apresentando as Tabelas de String 1313
Compreendendo os Recursos Personalizados 1314
Carregando Tabelas de String nos Programas com LoadString . . 1315
Listando o Conteúdo de um Arquivo de Recurso 1316
Usando EnumResourceTypes com Arquivos de Recursos . . . 1317
Carregando Recursos em Programas com FindResource . . . 1318

CAIXAS DE DIÁLOGO

Compreendendo as Caixas de Diálogo 1319
Definindo os Tipos de Diálogo 1320
Usando o Teclado com Caixas de Diálogo 1321
Compreendendo os Componentes do Gabarito da Caixa de Diálogo 1322
Criando um Gabarito de Caixa de Diálogo Específico . . . 1323
Compreendendo os Componentes da Definição da Caixa de Diálogo 1324
Definindo os Controles da Caixa de Diálogo 1325
Usando a Macro DialogBox para Exibir uma Caixa de Diálogo 1326
Compreendendo o Laço de Mensagem da Caixa de Diálogo . . 1327
Mais sobre a Manipulação de Controles 1328
Compreendendo a Macro CreateDialog 1329
Compreendendo a Função CreateDialogParam 1330
Processamento de Mensagem Padrão Dentro de uma Caixa de Diálogo 1331
Usando a Função DlgDirList para Criar uma Caixa de Lista de Diálogo 1332
Respondendo às Seleções do Usuário Dentro da Caixa de Lista 1333
Fechando a Caixa de Diálogo 1334
Compreendendo a Entrada do Usuário 1335
Respondendo aos Eventos do Mouse 1336
Usando a Mensagem WM_MOUSEMOVE 1337
Lendo os Botões do Mouse 1338
Respondendo aos Eventos do Teclado 1339
Compreendendo as Teclas Virtuais 1340
Usando as Teclas Virtuais 1341
Mais sobre o Uso da Mensagem WM_KEYDOWN . . 1342
Definindo e Retornando o Tempo do Clique Duplo no Mouse 1343
Permutando os Botões do Mouse 1344
Determinando se o Usuário Pressionou uma Tecla 1345
Apresentando as Barras de Rolagem 1346
Compreendendo os Diferentes Tipos de Barra de Rolagem . 1347
Usando a Função ShowScrollBar 1348
Compreendendo a Posição e a Faixa da Barra de Rolagem . . 1349

Compreendendo as Mensagens da Barra de Rolagem . . . 1350
Obtendo as Definições Atuais da Barra de Rolagem . 1351
Rolando o Conteúdo da Janela 1352
Compreendendo a Mensagem WM_SIZE . 1353
Compreendendo a Mensagem WM_PAINT 1354
Outras Mensagens da Barra de Rolagem Que Seus Programas Podem Capturar 1355
Habilitando e Desabilitando as Barras de Rolagem . . . 1356
Usando a Função ScrollDC 1357

GERENCIAMENTO DA MEMÓRIA NO WINDOWS

Compreendendo o Modelo de Memória Win32 1358
Compreendendo a Memória Global e a Local . . 1359
Compreendendo a Memória Virtual 1360
Revisitando os Heaps 1361
Alocando um Bloco de Memória a partir do Heap Global . . . 1362
Usando GlobalRealloc para Alterar Dinamicamente os Tamanhos do Heap 1363
Descartando um Bloco de Memória Alocado 1364
Usando a Função GlobalFree 1365
Usando GlobalLock e GlobalHandle 1366
Verificando a Memória do Computador 1367
Criando um Heap Dentro de um Processo 1368
Usando as Funções do Heap para Gerenciar a Memória Específica do Processo 1369
Verificando o Tamanho da Memória Alocada a partir de um Heap 1370
Alocando um Bloco de Memória Virtual 1371
Compreendendo as Páginas de Guarda 1372
Compreendendo Melhor os Blocos de Memória Virtuais 1373
Liberando Memória Virtual 1374
Gerenciando as Páginas de Memória Virtual 1375

PROCESSOS E ENCADEAMENTOS

Compreendendo Melhor os Processos 1376
Criando um Processo 1377
Terminando os Processos 1378
Gerando Processos-filho 1379
Trabalhando Mais com os Processos-filho 1380
Rodando Processos filho Destacados 1381
Compreendendo Melhor os Encadeamentos 1382
Avaliando a Necessidade de Encadeamentos 1383
Determinando Quando Não Criar um Encadeamento . . . 1384
Criando uma Função Simples de Encadeamento 1385
Visualizando a Iniciação do Encadeamento 1386
Passos Que o Sistema Operacional Efetua na Criação do Encadeamento 1387
Determinando o Tamanho da Pilha do Encadeamento 1388
Obtendo um Indicativo para o Processo ou o Encadeamento Atual 1389
Tratando o Tempo de Processamento do Encadeamento 1390
Gerenciando o Tempo de Processamento de Múltiplos Encadeamentos 1391
Compreendendo Melhor a Função GetQueueStatus 1392
Tratando as Exceções Não-tratadas 1393
Terminando os Encadeamentos 1394
Determinando a Identificação de um Encadeamento ou de um Processo 1395
Compreendendo Como o Sistema Operacional Escalona os Encadeamentos 1396
Apresentando os Níveis de Prioridade 1397
Compreendendo as Classes de Prioridade do Windows . . . 1398
Alterando a Classe de Prioridade de um Processo 1399
Definindo a Prioridade Relativa de um Encadeamento 1400
Obtendo o Nível de Prioridade Atual de um Encadeamento . . . 1401
Obtendo o Contexto de um Encadeamento 1402
Fazendo uma Pausa e Reiniciando os Encadeamentos 1403
Compreendendo a Sincronização de Encadeamentos 1404
Definindo os Cinco Principais Objetos de Sincronização . . . 1405
Criando uma Seção Crítica 1406
Usando uma Seção Crítica Simples 1407
Usando WaitForSingleObject para Sincronizar Dois Encadeamentos 1408
Usando WaitForMultipleObject para Sincronizar Muitos Encadeamentos 1409
Criando um Mutex 1410
Usando um Mutex Dentro de um Programa de Exemplo . 1411
Usando Semáforos 1412
Usando um Processador de Evento Simples 1413
Compreendendo a Interface de Dispositivo Gráfico 1414
Razões para Usar a Interface de Dispositivo Gráfico 1415
Compreendendo Melhor os Dispositivos do Contexto 1416
Usando Dispositivos do Contexto Privados 1417
Compreendendo as Origens e as Extensões 1418
Obtendo o Dispositivo de Contexto para uma Janela 1419
Criando um Dispositivo do Contexto para uma Impressora . . 1420
Usando CreateCompatibleDC para Criar um Dispositivo do Contexto em Memória 1421
Compreendendo os Perigos de CreateDC 1422
Usando a Função CreateFont 1423
Usando a Função EnumFontFamilies 1424
Exibindo Múltiplas Fontes com CreateFontIndirect . . . 1425
Recuperando as Capacidades de um Dispositivo 1426
Usando a Função GetSystemMetrics para Analisar uma Janela 1427
Compreendendo os Usos para GetSystemMetrics 1428
Obtendo um Dispositivo do Contexto para uma Janela Inteira 1429
Liberando os Dispositivos do Contexto 1430

BITMAPS, META-ARQUIVOS E ÍCONES

Obtendo um Indicativo de uma Janela a partir do Dispositivo do Contexto 1431
Compreendendo os Mapas de Bits Dependentes do Dispositivo 1432
Compreendendo os Mapas de Bits Independentes do Dispositivo 1433
Criando Mapas de Bits 1434
Exibindo Mapas de Bits 1435
Criando Mapas de Bits na Memória 1436
Preenchendo um Retângulo com um Padrão 1437
Usando SetDIBits 1438
Usando SetDIBitsToDevice para Exibir um Mapa de Bits em um Determinado Dispositivo 1439
Compreendendo os Meta-arquivos 1440
Criando e Exibindo Meta-arquivos 1441
Enumerando os Meta-arquivos Expandidos 1442
Usando a Função GetWinMetaFileBits 1443
Compreendendo os Ícones 1444
Criando Ícones 1445
Criando Ícones a partir de um Recurso 1446
Usando a Função CreateIconIndirect 1447
Usando a Função LoadIcon 1448
Usando LoadImage para Carregar Múltiplos Tipos Gráficos . . . 1449

E/S NO WINDOWS

Compreendendo a E/S em Arquivos no Windows . 1450
Apresentando as Canalizações, Recursos, Dispositivos e Arquivos 1451
Usando a Função CreateFile para Abrir Arquivos 1452
Usando CreateFile com Diferentes Dispositivos 1453
Usando Indicativos de Arquivo 1454
Revisitando os Ponteiros de Arquivo 1455
Usando WriteFile para Escrever em um Arquivo 1456
Usando ReadFile para Ler a partir de um Arquivo . . . 1457
Fechando o Arquivo 1458
Compartilhando Dados com o Mapeamento de Arquivos . . . 1459
Mapeando um Arquivo com a Memória Virtual 1460
Mapeando uma Exibição de um Arquivo no Processo Atual . 1461
Abrindo um Objeto de Mapeamento de Arquivo Nomeado . . 1462

Compreendendo os Atributos de Arquivo 1463
Obtendo e Alterando os Atributos de um Arquivo 1464
Obtendo o Tamanho de um Arquivo 1465
Obtendo a Data e a Hora de um Arquivo 1466
Criando Diretórios 1467
Obtendo e Definindo o Diretório Atual 1468
Obtendo os Diretórios Windows e System 1469
Removendo Diretórios 1470
Copiando Arquivos 1471
Movendo e Renomeando Arquivos 1472
Apagando Arquivos 1473
Usando FindFirstFile para Localizar Arquivos 1474
Usando FindNextFile 1475
Fechando o Indicativo de Pesquisa com FindClose 1476
Pesquisando por Atributos com as Funções FindFile 1477
Usando SearchPath em Vez de Find para Localizar 1478
Obtendo um Caminho Temporário 1479
Criando Arquivos Temporários 1480
Introduzindo a Função CreateNamedPipe 1481
Conectando uma Canalização Nomeada 1482
Chamando uma Canalização Nomeada 1483
Desconectando uma Canalização Nomeada 1484
Compreendendo Melhor o Processamento Assíncrono 1485
Usando Entrada e Saída Assíncrona 1486
Compreendendo a Estrutura OVERLAPPED 1487
E/S Assíncrona com um Objeto de Núcleo do Dispositivo 1488
Compreendendo as Quotas de Tamanho do Working Set 1489
Definindo Quotas Maiores e Menores 1490
Compreendendo a Função GetLastError 1491
Formatando as Mensagens de Erro com FormatMessage 1492
E/S Assíncrona com um Objeto Evento do Núcleo 1493
Usando WaitForMultipleObjects com E/S Assíncrona 1494
Apresentando as Portas de Finalização de E/S 1495
Usando E/S Alertável Para o Processamento Assíncrono 1496
A E/S Alertável Somente Funciona no Windows NT 1497
Usando ReadFileEx e WriteFileEx 1498
Usando uma Rotina de Callback de Finalização 1499
Usando um Programa de E/S Alertável 1500

SOBRE O CD-ROM

Além do código do programa para todas as dicas deste livro, o CD-ROM que acompanha este livro também inclui o compilador Borland *Turbo C++ Lite*, que você pode usar para começar a escrever programas C++ imediatamente.

TURBO C++ LITE

Para lhe ajudar a iniciar com a programação em C++, a Borland International permitiu que incluíssemos o compilador *Turbo C++ Lite* no CD-ROM.

Como você aprenderá, esse compilador é um programa baseado no MS-DOS. Se você estiver usando o Windows, poderá rodar o compilador *Turbo C++ Lite* dentro de uma janela do DOS.

O *Turbo C++ Lite* é um compilador completo que você pode usar para começar a escrever programas C e C++. Na verdade, você terá condições de escrever seu primeiro programa 10 minutos depois de começar a ler este livro. O *Turbo C++ Lite* é um compilador baseado no MS-DOS que lhe permite:

- Criar e editar seus programas C++, salvando os comandos em um arquivo-fonte no disco.
- Compilar os comandos do seu programa e gerar um arquivo executável.
- Compilar a maior parte dos programas que este livro apresenta em suas primeiras 1200 dicas.

O CD-ROM que acompanha este livro inclui instruções completas sobre como instalar e começar a usar o compilador *Turbo C++ Lite*, esteja você usando o Windows 3.1, Windows 95/98 ou o Windows NT.

ARQUIVOS-FONTE EM C E C++

O CD-ROM inclui também o código-fonte para todos os programas apresentados ou referenciados no livro. O disco está dividido em 15 diretórios de código, cada um na forma DicaXXXX, em que XXXX representa o número da última dica apresentada dentro do diretório. Além disso, cada diretório está dividido em um subdiretório para cada dica para que você possa fácil e rapidamente localizar o código-fonte para qualquer dica.

Para os programas referenciados mas não apresentados no livro, o código-fonte contém comentários para "conduzi-lo" ao processamento de cada programa. Muitos diretórios de dicas também incluem arquivos MAKE para simplificar seu acesso ao conteúdo do programa dentro desse diretório.

UMA INTRODUÇÃO À PROGRAMAÇÃO 1

Os programas de computador, também conhecidos como *software*, são formados por uma série de instruções que o computador executa. Quando você cria um programa, precisa especificar as instruções que o computador precisará executar para realizar as operações desejadas. O processo de definir as instruções que o computador deve executar é conhecido como *programação*. Ao criar um programa, você armazena as instruções em um arquivo ASCII cujo nome normalmente contém a extensão C para um programa C, e CPP para um programa C++. Por exemplo, caso você crie um programa C que executa operações de folha de pagamento, poderia chamar o arquivo que contém as instruções do programa de *folha.c*. Quando você cria programas, especifica as instruções desejadas usando uma *linguagem de programação*. C e C++ são somente duas dentre as muitas linguagens de programação que existem. Muitos programadores usam linguagens de programação, tais como BASIC, Pascal e FORTRAN. Cada linguagem de programação oferece certos recursos exclusivos, e tem seus pontos fortes e fracos. Em todos os casos, as linguagens de programação existem para permitir que definamos as instruções que queremos que o computador execute.

As instruções que o computador executa são, na verdade, séries de 1s e 0s, (dígitos binários) que representam os sinais eletrônicos que ocorrem dentro do computador. Para programar os primeiros computadores (nos anos 40 e 50), os programadores tinham que compreender como o computador interpretava diferentes combinações de 1s e 0s pois os programadores escreviam seus programas usando dígitos binários. À medida que os programas se tornaram maiores, tornou-se inviável trabalhar em termos de 1s e 0s do computador. Em vez disso, os pesquisadores criaram as linguagens de programação, que permitem que as pessoas expressem as instruções do computador de uma forma mais clara para os humanos. Após os programadores colocarem suas instruções em um arquivo (chamado *arquivo-fonte*), um segundo programa (chamado *compilador*) converte as instruções da linguagem de programação em 1s e 0s (conhecidas como *linguagem de máquina*) que o computador compreende. Os arquivos no seu disco com as extensões EXE e COM contêm o código de máquina (1s e 0s) que o computador executará. A Figura 1 ilustra o processo de compilar um arquivo de código-fonte em um programa executável.

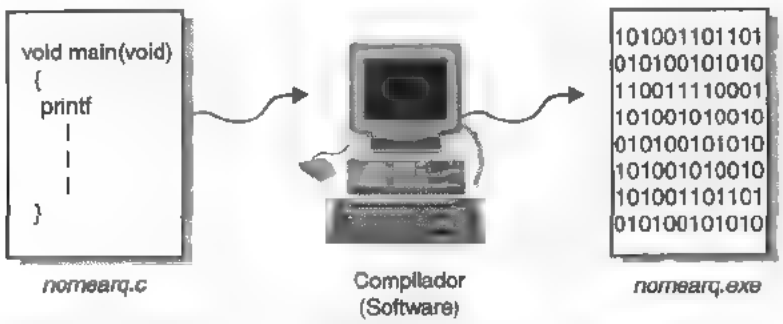

Figura 1[1] Um compilador converte instruções de código-fonte em código de máquina.

Após você criar um arquivo em código-fonte, roda um compilador para converter as instruções em um formato que o computador possa executar. Por exemplo, se você estiver usando o Borland *Turbo C++ Lite*™ (incluído no CD-ROM que acompanha este livro), você usará a opção Compile to OBJ do menu Compile para chamar o compilador (isto é, instruí-lo a compilar o código-fonte). As dicas a seguir lhe conduzirão nos passos necessários para criar e compilar um programa em C.

CRIANDO UM ARQUIVO-FONTE EM ASCII 2

Quando você cria um programa, precisa colocar os comandos de programa que você quer que o computador execute em um arquivo chamado *arquivo-fonte*. Se você não estiver usando o *Turbo C++ Lite* ou um compilador completo com editor, deverá criar seus arquivos de programas usando um editor ASCII, tal qual o programa EDIT, fornecido junto com o DOS. Você não deverá criar os programas usando um editor de textos (tal qual o Microsoft *Word*®, ou o *WordPerfect*® da Corel). Como você sabe, os processadores de texto permitem que você formate os documentos atribuindo margens, italizando ou sublinhando as palavras, e assim por diante. Para executar essas operações, os processadores de texto incorporam caracteres especiais dentro de seus documentos. Embora esses caracteres sejam compreensíveis para o processador de texto, eles confundirão o compilador que

1. As figuras e tabelas deste livro estão numeradas de acordo com o número da dica. Não segue, portanto, uma seqüência de numeração.

converte, o seu arquivo-fonte para código de máquina e essa confusão provocará erros. Quando você criar seu arquivo-fonte, certifique-se de atribuir um nome significativo que descreva com exatidão a função do programa para o arquivo. Por exemplo, você poderia nomear o código-fonte para um programa de faturamento como *fatura.c*, e o arquivo-fonte para um programa de jogo como *futebol.c*.

Por outro lado, se você estiver usando um compilador que inclua um editor interno, deverá criar seus programas dentro desse editor. Por exemplo, se você estiver usando o *Turbo C++ Lite*, usará a opção New do menu File para criar um novo arquivo de programa. Para criar seu primeiro programa dentro do *Turbo C++ Lite*, acompanhe os seguintes passos:

1. Selecione a opção New no menu File. O *Turbo C++ Lite* criará o arquivo *noname00.cpp*.
2. Digite o seguinte código na janela *noname00.cpp*:

```
#include <stdio.h>

void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

3. Selecione a opção Save As no menu File. O *Turbo C++ Lite* exibirá a caixa de diálogo Save File As.
4. Dentro da caixa de diálogo Save File As, digite o nome *primeiro.c* e pressione Enter. O *Turbo C++ Lite* salvará o arquivo de programa *primeiro.c*.

Embora o programa *primeiro.c* contenha seis linhas, somente o comando *printf* na verdade executa algum trabalho. Quando você executar esse programa, *printf* exibirá a mensagem *Bíblia do Programador C/C++, do Jamsa!* na sua tela. Toda linguagem de programação (exatamente como as linguagens humanas, tais como português, inglês, francês e alemão) tem um conjunto de regras, chamadas *regras de sintaxe*, às quais você precisa obedecer quando usar a linguagem. Ao criar programas C, você deve obedecer às regras de sintaxe da linguagem C. Exemplos de regras de sintaxe incluem os parênteses que seguem o nome *main* e o ponto-e-vírgula no final da instrução *printf*. Quando você digitar seu programa, precisa ter muito cuidado para não omitir algum desses elementos. Confira sua digitação para garantir que você tenha digitado as instruções do programa C exatamente como já mostrado. Se as instruções estiverem corretas, salve o conteúdo do arquivo no seu disco. Na próxima dica, você aprenderá a compilar seu arquivo-fonte e a converter suas instruções de programação C na linguagem de máquina que seu computador pode compreender e executar.

3 COMPILANDO SEU PROGRAMA

Na dica anterior, você criou o arquivo-fonte, *primeiro.c*, que contém o comando *printf* que exibirá a mensagem *Bíblia do Programador C/C++, do Jamsa!* na sua tela quando você executar o programa. Um arquivo-fonte contém instruções em um formato que você pode compreender (ou pelo menos poderá compreender após aprender C). Um programa executável, por outro lado, contém instruções expressas como 1s e 0s que o computador compreende. O processo de converter seu arquivo-fonte C em código de máquina é chamado *compilação*. Dependendo do compilador C que você estiver usando, diferirá o comando que você precisa executar para compilar seu arquivo-fonte. Assumindo que você esteja usando o *Turbo C++ Lite*, da Borland, é possível compilar o programa (*primeiro.c*) que você criou na Dica 2, usando a seguinte seqüência de comandos:

1. Selecione a opção Build All no menu Compile. O *Turbo C++ Lite* exibirá a caixa de diálogo Compiling.
2. Se o compilador completar com sucesso a compilação, ele pedirá que você pressione qualquer tecla. Se o compilador não criar o arquivo *primeiro.exe*, mas, em vez disso exibir mensagens de erro na sua tela, provavelmente é porque você violou uma regra de sintaxe de C, como discutido na próxima dica.
3. Se você digitou sem erros os comandos C, como mostrado na Dica 2, o compilador C criará um arquivo executável chamado *primeiro.exe*. Para executar o programa primeiro.exe, você pode selecionar a opção Run no menu Run ou pressionar o atalho de teclado Ctrl+F9.

Quando você executar o programa, sua tela exibirá a seguinte saída:

```
Bíblia do Programador C/C++, do Jamsa!
C:\>
```

Nota: Em algumas instalações, o Turbo C++ Lite gerará a saída e retornará imediatamente à janela de edição. Nesses casos, selecione a opção DOS Shell no menu File para ver a saída gerada pelo programa.

COMPREENDENDO OS ERROS DE SINTAXE 4

Como você leu na Dica 2, toda linguagem de programação tem um conjunto de regras, chamadas *regras de sintaxe*, às quais você precisa obedecer quando especificar os comandos do programa. Se você violar uma regra de sintaxe, seu programa não será compilado com sucesso. Em vez disso, o compilador exibirá mensagens de erro na sua tela que especificam a linha do seu programa que contém o erro e uma breve descrição do erro. Usando seu editor, crie o arquivo *sintaxe.c*, que contém um erro de sintaxe. No exemplo a seguir, o programa deixou de incluir as aspas finais na mensagem *Bíblia do Programador C/C++*, do Jamsa!:

```
#include <stdio.h>

void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!);
 }
```

Quando você compilar este programa, seu compilador exibirá uma mensagem de erro de sintaxe ao encontrar a linha 5. Dependendo do seu compilador, a mensagem de erro real diferirá. No caso do *Turbo C++ Lite*, sua tela exibirá as seguintes mensagens:

```
Error sintaxe.c 5:  Unterminated string or character constant in function main
Error sintaxe.c 6:  Function call missing ) in function main()
Error sintaxe.c 6:  Statement missing ; in function main()
```

Embora o código-fonte *sintaxe.c* contenha somente um erro, o compilador C exibirá três mensagens de erros. A falta das aspas finais provoca uma série de erros em cascata (um erro que leva a outro) dentro do compilador.

Para corrigir os erros de sintaxe dentro de seus programas, acompanhe os seguintes passos:

1. Escreva o número da linha de cada erro e uma breve descrição.
2. Edite seu arquivo-fonte, movendo o cursor para o número da primeira linha que o compilador exibir.
3. Dentro do arquivo-fonte, corrija o erro e mova o cursor para o número da próxima linha. A maioria dos editores exibe o número da linha atual para lhe ajudar a localizar linhas específicas dentro do arquivo.

No caso do arquivo *sintaxe.c*, edite o arquivo e acrescente as aspas que faltaram. Salve o arquivo no disco e compile-o novamente. Após você corrigir o erro de sintaxe, o compilador criará o arquivo *sintaxe.c*. Para executar *sintaxe.exe*, selecione a opção Run no menu Run. O programa executará e produzirá a saída mostrada aqui:

```
A Bíblia do Programador C/C++ - A Bíblia, do Jamsa!
C:\>
```

ESTRUTURA DE UM TÍPICO PROGRAMA EM C 5

Na Dica 2, você criou o arquivo-fonte *primeiro.c*, que continha os seguintes comandos:

```
#include <stdio.h>

void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

Esses comandos são similares aos que você encontrará na maioria dos programas C. Em muitos casos, um arquivo-fonte C pode começar com um ou mais comandos *#include*. O comando *#include* instrui o compilador C a usar o conteúdo de um arquivo específico.

No caso do arquivo *primeiro.c*, o comando *#include* instrui o compilador C a usar um arquivo chamado *stdio.h*. Os arquivos especificados por um comando *#include* são arquivos ASCII, que contêm código-fonte em C. Você pode imprimir ou exibir o conteúdo de cada arquivo seguindo os passos discutidos na Dica 13, mais à frente. Os arquivos que você nomeia dentro de um comando *#include*, os quais normalmente usam a extensão *h*, são chamados *arquivos de inclusão* ou *arquivos de cabeçalho*. A maioria dos arquivos de cabeçalho contém comandos que seus programas usam comumente, embora você aprenderá mais tarde, neste livro, sobre outros usos para os arquivos de cabeçalho. Quando você instrui o compilador C a incluir o conteúdo do arquivo, não precisa digitar os comandos nos seus programas. Após os comandos *#include*, você normalmente encontrará um comando similar ao seguinte:

```
void main(void)
```

Cada programa C que você criar incluirá uma linha similar ao comando *void main*. Como já foi escrito na Dica 1, um programa C contém uma lista de instruções que você quer que o compilador execute. À medida que a complexidade de seus programas aumentar, divida-os em partes menores que sejam mais fáceis para você (e para outros que vierem a ler seu programa) compreender. O grupo de instruções que você quer que o programa execute primeiro chama-se *programa main* (principal). O comando *void main* identifica esses comandos (o programa main) para o compilador.

É claro que, como o compilador C determinará quais comandos formam o programa main e quais comandos são suplementares, você precisa ter um modo de dizer ao compilador C quais instruções correspondem a cada seção do seu programa. Para atribuir comandos do programa a uma seção específica do seu programa, coloque o comando dentro de abre e fecha chaves ({ e }). As chaves fazem parte da sintaxe de C. Para todo abre chaves, você precisa ter um fecha chaves correspondente que encerra o grupo de comandos

6 ACRESCENTANDO COMANDOS AOS SEUS PROGRAMAS

Como você leu, o programa *primeiro.c* usou o comando *printf* para exibir uma mensagem na sua tela. O programa a seguir, *3_msgs.c*, usa três comandos *printf* para exibir a mesma mensagem. Cada comando está contido dentro do abre e fecha chaves:

```
#include <stdio.h>

void main(void)
{
   printf("Programando");
   printf("em C/C++, ");
   printf("a Bíblia!");
}
```

Observe o caractere de espaço em branco dentro dos comandos *printf*. O caractere de espaço é importante porque garante que o programa exibirá corretamente o texto na sua tela (colocando um espaço entre as palavras). À medida que o número de comandos nos seus programas aumentar, assim também aumentará a probabilidade de erros de sintaxe. Confira seu programa para garantir que você digitou corretamente cada comando e depois salve o arquivo no disco. Quando você compilar e executar o programa *3_msgs.c*, sua tela exibirá a seguinte saída:

```
Bíblia do Programador C/C++
C:\>
```

7 EXIBINDO SAÍDA EM UMA NOVA LINHA

Vários dos programas anteriores exibiram a mensagem *Bíblia do Programador C/C++*, do Jamsa! na tela. À medida que seus programas se tornarem mais complexos, você pode querer que eles exibam sua saída em duas ou mais linhas. Na Dica 6, você criou o programa *3_msgs.c*, que usou três comandos *printf* para exibir uma mensagem na tela.

```
   printf("Bíblia");
   printf("do Programador C/C++, ");
   printf("do Jamsa!");
```

A não ser que você instrua *printf* a fazer de forma diferente, *printf* continuará sua saída na linha atual. O objetivo do programa a seguir, *uma_lin.c*, é exibir saída em duas linhas sucessivas:

```
#include <stdio.h>

void main(void)
 {
   printf ("Esta é a linha um.");
   printf ("Esta é a linha dois.");
 }
```

Quando você compilar e executar o programa *uma_lin.c*, sua tela exibirá a seguinte saída:

```
Esta é a linha um.Esta é a linha dois.
C:\>
```

Quando você quiser que *printf* inicie sua saída em uma nova linha, é preciso incluir o *caractere de nova linha* especial (\n) dentro do texto que você pede para *printf* exibir. Quando *printf* encontrar o caractere \n, avançará o cursor para o início da próxima linha. O programa a seguir, *duas_lin.c*, usa o caractere de nova linha para exibir a segunda linha de texto em uma nova linha, como desejado:

```
#include <stdio.h>

void main(void)
 {
   printf ("Esta é a linha um.\n");
   printf ("Esta é a linha dois.");
 }
```

Quando você compilar e executar o programa *duas_lin.c*, sua tela exibirá a seguinte saída:

```
Esta é a linha um.
Esta é a linha dois.
C:\>
```

Muitos dos programas apresentados neste livro usam o caractere de nova linha. Na verdade, quase todo programa que você escrever normalmente usará o caractere de nova linha em um ou mais lugares.

CONSIDERA AS LETRAS MAIÚSCULAS E MINÚSCULAS DIFERENTES 8

À medida que você vai digitando seus programas, lembre-se de que C considera as letras maiúsculas e minúsculas como diferentes. Como regra, a maioria dos comandos C usa minúsculas, a maioria das constantes C usa todas maiúsculas; e quase todas as variáveis C misturam a caixa das letras. Os programas C fazem muito uso de letras minúsculas. Como o programa a seguir, *erro_mai.c*, usa a letra M maiúscula no nome *Main* quando C espera o nome *main*, o programa não passa pela compilação.

```
#include <stdio.h>

void Main(void)
 {
   printf ("Este programa não passa na compilação.");
 }
```

Quando você compilar o programa *erro_mai.c*, o compilador *Turbo C++ Lite* exibirá a seguinte mensagem:

```
Linker error: Undefined symbol _main in module TURBO C\COS.ASM
```

Essa mensagem quase incompreensível que o compilador *Turbo C++ Lite* exibe é consequência do erro na caixa da letra M na palavra *Main*. Para corrigir o erro, você simplesmente altera *Main* para *main*. Após fazer a correção, recompile e execute o programa.

9 COMPREENDENDO OS ERROS LÓGICOS

Na Dica 4, você aprendeu que caso viole uma das regras da linguagem C, o compilador exibirá uma mensagem de erro de sintaxe e seu programa não será compilado com sucesso. À medida que seus programas se tornarem mais complexos, algumas vezes eles passarão na compilação mas sem realizar corretamente a tarefa que você queria que eles executassem. Por exemplo, assuma que você quer que o programa a seguir, *uma_lin.C*, exiba sua saída em duas linhas:

```
#include <stdio.h>

void main(void)
 {
   printf ("Esta é a linha um.");
   printf ("Esta é a linha dois.");
 }
```

Como o programa não viola nenhuma das regras de sintaxe de C, ele será compilado sem erros. No entanto, quando você executar o programa, ele não exibirá a saída em duas linhas; em vez disso, exibirá a saída em uma única linha, como mostrado aqui:

```
Esta é a linha um.Esta é a linha dois.
C:\>
```

Quando seu programa não funciona como você deseja, é porque ele contém *erros lógicos*, ou *bugs*. Quando seu programa contiver um erro lógico (e algum dia isso ocorrerá), será preciso descobrir e corrigir a causa do erro. O processo de remover os erros lógicos do seu programa é chamado *depuração*. Posteriormente, neste livro, você aprenderá várias técnicas diferentes que poderá usar para localizar os erros lógicos dentro de seu programa. Por enquanto, porém, o melhor modo de localizar esses erros é imprimir uma cópia do seu programa e examiná-la linha por linha até localizar o erro. O exame linha a linha de um programa é chamado *verificação de mesa*. No caso do programa *uma_lin.c*, sua verificação de mesa deve revelar que o primeiro comando *printf* não contém o caractere de nova linha (\n).

10 COMPREENDENDO O PROCESSO DE DESENVOLVIMENTO DE UM PROGRAMA

Quando você cria programas, normalmente segue os mesmos passos. Para começar, use um editor para criar seu arquivo-fonte. Em seguida, compile o programa. Se o programa contiver erros de sintaxe, você precisará editar o arquivo-fonte e corrigir os erros. Após o programa passar pela compilação sem apresentar erros, tente executá-lo. Se o programa rodar com sucesso e operar como você espera, o processo de criação do programa estará terminado. Por outro lado, se o programa não funcionar como planejado, você precisará analisar o código-fonte e tentar localizar o erro lógico (como discutido na Dica 9). Após você corrigir o erro, precisará compilar o código-fonte para criar um novo arquivo executável. Você poderá então testar o novo programa para se certificar de que ele executará a tarefa desejada. A Figura 10 ilustra o processo de desenvolvimento de um programa.

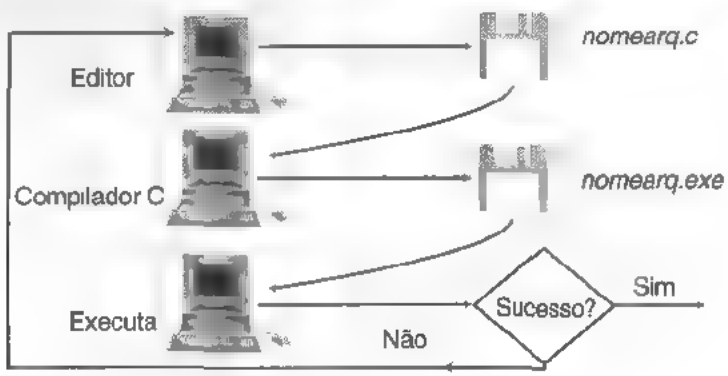

Figura 10 O processo de desenvolvimento de um programa.

COMPREENDENDO OS TIPOS DE ARQUIVO 11

Ao criar um programa C, você coloca seus comandos em um arquivo-fonte que tem a extensão C. Se seu programa for compilado com sucesso, o compilador criará um arquivo de programa executável com a extensão EXE. Como você já leu na Dica 5, muitos programas usam arquivos de cabeçalho (que têm a extensão *.h*) que contêm comandos usados comumente. Se você examinar seu diretório após compilar um programa, provavelmente encontrará um ou mais arquivos com a extensão *.OBJ*. Esses arquivos, chamados *arquivos-objetos*, contêm instruções na forma de 1s e 0s que o computador compreende. No entanto, você não pode executar esses arquivos, pois o conteúdo deles não está completo.

O compilador C fornece rotinas (tais como *printf*) que executam operações comumente usadas e reduzem o número de comandos que você precisa incluir nos seus programas. Após examinar a sintaxe do seu programa, o compilador criará um arquivo-objeto. No caso do programa *primeiro.c*, o compilador criaria um arquivo-objeto chamado *primeiro.obj*. Em seguida, um programa chamado *linkeditor* (ou *ligador*) combina os comandos do programa no seu arquivo-objeto com as funções (tais como *printf*) que o compilador fornece para criar o programa executável. Na maioria dos casos, quando você chama o compilador para examinar seu código-fonte, ele automaticamente chamará o *linkeditor* se o programa passar na compilação. A Figura 11 ilustra o processo de compilar e ligar um programa.

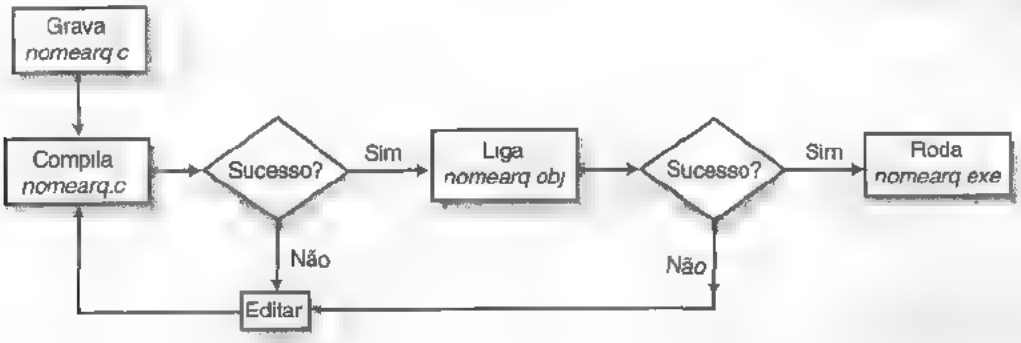

Figura 11 O processo de compilar e ligar um programa.

COMPREENDENDO MELHOR O LINKEDITOR 12

Na Dica 11 você aprendeu que quando compila seu programa C, um segundo programa, chamado linkeditor, combina os comandos do seu programa com rotinas predefinidas (que o compilador fornece) para converter um arquivo-objeto em um programa executável. Assim como o caso com o processo de compilação, que pode detectar erros de sintaxe, o processo de ligação também pode encontrar erros. Por exemplo, considere o seguinte programa, *sem_prin.c*, que erroneamente usa *print* em vez de *printf*:

```
#include <stdio.h>

void main(void)
 {
   print("Este programa não passa na ligação");
 }
```

Como o programa *sem_prin.c* não viola nenhuma regra de sintaxe de C, ele será compilado com sucesso, produzindo um arquivo-objeto. No entanto, o linkeditor do *Turbo C++ Lite* exibirá a seguinte mensagem de erro, pois o comando *print* não está definido:

```
Error: Function 'print' should have a prototype in function main()
```

Como o compilador C não fornece uma função chamada *print*, o linkeditor não pode criar o programa executável *sem_prin.c*. Em vez disso, o linkeditor exibirá a mensagem de erro mostrada anteriormente. Para solucionar o problema, edite o arquivo, mudando *print* para *printf*, e, depois, recompile e ligue seu programa.

13 COMPREENDENDO OS ARQUIVOS DE CABEÇALHO

Cada programa apresentado neste livro usa um ou mais comandos *#include* para instruir o compilador C a usar os comandos que um arquivo de cabeçalho contém. Um arquivo de cabeçalho é um arquivo ASCII cujo conteúdo você pode imprimir ou exibir na sua tela. Se você examinar o diretório que contém seu compilador (o diretório *tclite*, no caso do *Turbo C++ Lite*), encontrará um subdiretório chamado *include*. O subdiretório *include* contém os arquivos de cabeçalho do compilador. Separe um tempo para localizar os arquivos de cabeçalho do seu compilador. Se desejar, imprima o conteúdo de um arquivo de cabeçalho usado frequentemente, tal como *stdio.h*. Você encontrará comandos de programação C dentro de um arquivo de inclusão. Quando o compilador C encontra um comando *#include* no seu programa, ele compila o código que o cabeçalho contém exatamente como se você tivesse digitado o conteúdo do arquivo de cabeçalho no código-fonte do seu programa. Os arquivos de cabeçalho contêm definições comumente usadas e fornecem ao compilador informações sobre certas funções, tais como *printf*. Por ora, você pode achar o conteúdo de um arquivo de cabeçalho difícil de compreender. No entanto, à medida que se tornar mais fluente em C e C++, deverá imprimir uma cópia e examinar todo arquivo de cabeçalho que usar. Os arquivos de cabeçalho contêm informações valiosas e apresentam técnicas de programação que farão de você um programador C melhor.

14 AJUDANDO O COMPILADOR A LOCALIZAR OS ARQUIVOS DE CABEÇALHO

Na Dica 13 você aprendeu que, quando o compilador C encontra um comando *#include*, ele acrescenta o conteúdo do arquivo de cabeçalho ao seu programa, exatamente como se você tivesse digitado o conteúdo do arquivo de cabeçalho no seu código-fonte. Dependendo do seu compilador, as variáveis do ambiente podem conter uma linha INCLUDE que diz ao compilador o nome do subdiretório que contém os arquivos de cabeçalho. Se quando você compilar um programa seu compilador exibir uma mensagem de erro dizendo que não pode abrir um determinado arquivo de cabeçalho, verifique primeiro o subdiretório que contém os arquivos de cabeçalho de seu programa para garantir que o arquivo existe. Se você encontrar o arquivo, peça um comando SET no prompt do DOS, como mostrado aqui:

```
C:\> SET    <Enter>
COMSPEC=C:\DOS\COMMAND.COM
PATH=C:\DOS;C:\WINDOWS;C:\BORLANDC\BIN
PROMPT=$P$G
TEMP=C:\TEMP
```

Se o seu ambiente não contiver uma linha INCLUDE, verifique a documentação que acompanhou seu compilador para determinar se seu compilador requer essa linha. Normalmente, a instalação do compilador colocará dentro do arquivo *autoexec.bat* um comando SET que atribui a entrada INCLUDE ao subdiretório que contém os arquivos de cabeçalho, como mostrado aqui:

```
SET INCLUDE=C:\BORLANDC\INCLUDE
```

Se o seu compilador usa a entrada INCLUDE, e seu arquivo *autoexec.bat* não a define, você mesmo pode incluí-la no arquivo *autoexec.bat*.

Nota: O compilador Turbo C++ Lite procura os arquivos de inclusão somente no subdiretório include.

15 AGILIZANDO AS COMPILAÇÕES

Quando você compilar um arquivo-fonte, o compilador C poderá criar um ou mais arquivos temporários que existirão somente enquanto o compilador e linkeditor estiverem trabalhando. Dependendo do seu compilador, você poderá usar a variável do ambiente TEMP para especificar onde o compilador criará esses arquivos temporários. Se o seu computador tiver vários discos rígidos, alguns dos quais com mais espaço disponível que os outros (especialmente se seu compilador rodar dentro do Windows e, portanto, usar memória virtual e arquivos de transferência), você poderá fazer a variável TEMP apontar para o disco que tiver mais espaço disponível. Desse modo, o compilador criará seus arquivos temporários no disco rígido mais rápido, o que agilizará o processo de

compilação. Assumindo que sua unidade D tenha esse espaço extra, você poderá colocar um comando SET dentro de seu *autoexec bat* para atribuir a variável TEMP à unidade *D*, como mostrado aqui:

```
SET TEMP=D:
```

COMENTANDO SEUS PROGRAMAS 16

Como regra, cada vez que você cria um programa, precisa incluir comentários que expliquem o processamento que o programa realiza. Basicamente, um comentário é uma mensagem que o ajuda a ler e compreender o programa. À medida que seus programas aumentarem de tamanho, eles se tornarão mais difíceis de compreender. Como você criará centenas e talvez até milhares de programas, não conseguirá lembrar o propósito de cada comando dentro de cada programa. Se você incluir comentários no seu programa, não precisará lembrar os detalhes de cada programa. Em vez disso, os comentários explicarão o processamento.

A maioria dos compiladores C e C++ oferece dois modos para você colocar comentários dentro do seu arquivo-fonte. Primeiro, você coloca dois caracteres de barras inclinadas para a direita (//), como mostrado aqui:

```
// Isto é um comentário
```

Quando o compilador C encontra as barras duplas, ele ignora o texto que segue até o final da linha atual. O programa a seguir, *comenta.c*, ilustra o uso de comentários:

```
// Programa: comenta.c
// Escrito por: Kris Jamsa e Lars Klander
// Data de criação: 22-12-97
// Propósito: Ilustrar o uso de comentários em um programa C.

#include <stdio.h>

void main(void)
 {
   printf ("Bíblia do Programador C/C++, do Jamsa!"); // Exibe uma mensagem
 }
```

Neste exemplo, você imediatamente sabe, lendo esses simples comentários, quando, por que e quem escreveu o programa. Adquira o hábito de colocar comentários similares no início dos seus programas. Se outros programadores que precisarem ler ou modificar o programa tiverem dúvidas, eles saberão rapidamente quem foi o autor original do programa.

Quando o compilador C encontra as barras duplas (//), ele ignora o texto até o final daquela linha. A maioria dos novos arquivos-fonte C usam as barras duplas para designar um comentário. Se você estiver lendo programas C antigos, poderá encontrar comentários escritos de uma segunda forma. Na segunda forma aceitável de comentários, esses aparecem entre um conjunto de barras e asteriscos, como mostrado aqui:

```
/* Isto é um comentário */
```

Quando o compilador encontra o símbolo de abertura de comentário (/*), ele ignora todo o texto até e incluindo o símbolo de encerramento de comentário (*/). Usando o formato */* comentário **/*, um único comentário pode aparecer em duas ou mais linhas. O programa a seguir, *comenta2.c*, ilustra o uso do formato */* comentário **/*

```
/* Programa: COMENTA.C
   Escrito por: Kris Jamsa e Lars Klander
   Data de criação: 22-12-97

   Propósito: Ilustrar o uso de comentário em um programa C. */

#include <stdio.h>

void main(void)
 {
```

```
    printf ("Bíblia do Programador C/C++, do Jamsa!"); /* Exibe uma mensagem */
}
```

Como você pode ver, o primeiro comentário do programa contém cinco linhas. Quando você usa o formato */* *comentário* */* para seus comentários, certifique-se de que cada símbolo de início de comentário (/*) tenha um símbolo de finalização correspondente (*/). Se o símbolo de finalização estiver faltando, o compilador C ignorará a maior parte do seu programa, o que, eventualmente, resultará em erros de sintaxe que serão difíceis de detectar.

A maioria dos compiladores C retornará um erro de sintaxe se você tentar colocar um comentário dentro de outro, como mostrado aqui:

```
/* Este comentário tem /* um segundo */ comentário dentro */
```

17 MELHORANDO A LEGIBILIDADE DO SEU PROGRAMA

Na Dica 16, você aprendeu como usar comentários dentro de seus programas para melhorar sua legibilidade. Toda vez que criar um programa, assuma que você ou outro programador terá que modificá-lo de alguma forma. Portanto, é essencial que você escreva seus programas de modo que eles sejam fáceis de ler. O programa C a seguir, *difícil.c*, exibirá uma mensagem na sua tela:

```
#include <stdio.h>
void main(void) {printf("Bíblia do Programador C/C++, do Jamsa!");}
```

Embora este programa seja compilado com sucesso e exiba a mensagem desejada, ele é difícil de ler, para dizer o mínimo. Um bom programa não somente funciona, mas também é fácil de ler e de compreender. A chave para criar programas legíveis é incluir comentários que explicam o processamento de programa e usar linhas em branco para melhorar o formato do programa. Em dicas posteriores você aprenderá o importante papel que as endentações (ou recuos) têm para produzir código legível de programas.

18 PRESTANDO ATENÇÃO ÀS MENSAGENS DE ADVERTÊNCIA DO COMPILADOR

Quando seu programa contém um ou mais erros de sintaxe, o compilador C exibirá mensagens de erro na sua tela e não criará um programa executável. À medida que você for criando programas, algumas vezes o compilador exibirá uma ou mais mensagens de advertência na sua tela, mas ainda criará o arquivo do programa executável. Por exemplo, o seguinte programa C, *semstdio.c*, não inclui o arquivo de cabeçalho *stdio.h*:

```
void main(void)
{
   printf("Bíblia do Programador C/C++, do Jamsa!");
}
```

Quando você compilar este programa, o compilador *Turbo C++ Lite* exibirá a seguinte mensagem de advertência:

```
Warning semstdio.c 3:    Function 'printf' should have a prototype in function
```

Quando o compilador exibir uma mensagem de advertência, você deverá determinar imediatamente a causa da queixa do compilador e corrigi-la. Embora as advertências possam nunca causar um erro durante a execução do seu programa, algumas advertências criarão a oportunidade para erros que são muito difíceis de depurar mais tarde. Separando tempo para localizar e corrigir a causa das advertências do compilador, você aprenderá muito mais sobre o funcionamento interno de C e C++.

19 CONTROLANDO AS ADVERTÊNCIAS DO COMPILADOR

Na Dica 18, você aprendeu que deve prestar atenção às mensagens de advertência que o compilador exibe na tela. Para lhe ajudar a fazer melhor uso das advertências do compilador, muitos compiladores permitem que você defina o nível de mensagem que deseja. Dependendo do seu compilador, você poderá usar uma opção de linha de comando para controlar o nível de advertência ou pode usar *pragmas*, explicadas na Dica 145. Uma *pragma*

é uma diretiva para o compilador. Como você aprenderá, diferentes compiladores suportam diferentes *pragmas*. Por exemplo, para desabilitar a advertência *Identifier is declared but never used* (O identificador é declarado mas nunca usado) dentro do *Turbo C++ Lite*, seu código incluiria a seguinte *pragma:*

```
#pragma warn -use
```

Se você não estiver usando o *Turbo C++ Lite*, consulte a documentação que acompanha seu compilador para determinar se você pode desativar mensagens de advertências específicas.

USANDO COMENTÁRIOS PARA EXCLUIR COMANDOS DO PROGRAMA 20

Na Dica 16, você aprendeu que deve usar comentários dentro de seus programas para melhorar a legibilidade. À medida que seus programas se tornarem mais complexos, você poderá usar comentários para ajudar a depurar (remover erros) de seus programas. Quando o compilador C encontrar as barras duplas (//), ele ignorará todo o texto restante até o fim da linha atual. Da mesma forma, quando o compilador encontrar o símbolo de início de comentário (/*), ele ignorará todo o texto que segue, até o símbolo de finalização de comentário (*/). Quando você testar seus programas, algumas vezes desejará eliminar um ou mais comandos do seu programa. Um modo de eliminar os comandos do seu programa é simplesmente excluir os comandos do seu arquivo-fonte. Um segundo modo de eliminar comandos é *transformá-los em comentários* No programa a seguir, *semsaida.c*, todos os comandos *printf* foram transformados em comentários:

```
#include <stdio.h>

void main(void)
 {
   // printf ("Esta linha não aparece");
    /* Isto é um comentário
       printf ("Esta linha também não aparece");
    */
 }
```

Como ambos os comandos *printf* aparecem dentro de comentários, o compilador ignora os dois Como resultado, nada aparece na tela quando você executa o programa À medida que seus programas se tornarem mais complexos, será muito conveniente poder usar comentários para desativar certos comandos.

Como você aprendeu na Dica 16, a maioria dos compiladores C retornará um ou mais erros de sintaxe se você tentar colocar um comentário dentro de outro. Quando você usar comentários para desabilitar comandos, tome cuidado para não colocar inadvertidamente um comentário dentro de outro

COMPREENDENDO A IMPORTÂNCIA DOS NOMES 21

À medida que você examinar as dicas apresentadas neste livro, encontrará nomes de variáveis e de funções cujos nomes iniciam com um traço de sublinhado, tal como *_dos_getdrive* ou *_chmode*. Em geral, você somente usa tais variáveis e funções dentro do ambiente do DOS. Se você estiver escrevendo programas que executarão sob o DOS, Windows, Macintosh, UNIX, ou possivelmente algum outro sistema operacional, deverá evitar usar essas funções, pois é provável que elas não estejam disponíveis nos outros sistemas. Portanto, para migrar seu programa do DOS para outro sistema operacional, você terá que fazer alguma programação adicional. Algumas funções podem ter duas implementações: uma com um traço de sublinhado *(_chmode)* e uma sem *(chmode)*. Como regra, use a função ou variável sem o sublinhado, que neste caso é *chmode*

COMPREENDENDO O PONTO-E-VÍRGULA 22

Examinando programas C, você verá que eles utilizam muito o caractere ponto-e-vírgula. O ponto-e-vírgula em C tem um significado especial. Como você sabe, um programa é uma lista de instruções que você quer que o computador execute. Ao especificar essas instruções em C, você usará o ponto-e-vírgula para separar um coman-

do do outro. À medida que seus programas vão se tornando mais complexos, poderá acontecer de um comando não caber em uma única linha. Quando o compilador C examinar seu programa, ele procurará o ponto-e-vírgula para distinguir um comando do próximo. A sintaxe da linguagem C define o uso do ponto-e-vírgula. Se você omitir o ponto-e-vírgula, ocorrerá um erro de sintaxe, e o programa não passará na compilação.

23 APRESENTANDO AS VARIÁVEIS

Para efetuar um trabalho útil, os programas precisam armazenar informações, tais como um documento editado em múltiplas seções, dentro de um arquivo e também internamente. Como você sabe, toda vez que roda um programa, o sistema operacional carrega as instruções do seu programa na memória do computador. À medida que o programa roda, ele armazena valores em posições de memória. Por exemplo, assuma que você tenha um programa que imprime um documento. Toda vez que você roda o programa, ele exibirá uma mensagem perguntando o nome do arquivo, bem como o número de cópias que você quer imprimir. Digitando essas informações, o programa armazena os valores digitados em posições de memória específicas. Para ajudar seu programa a controlar as posições de memória onde ele colocou dados, cada posição tem um *endereço* exclusivo, tal como endereço 0, 1, 2, 3 etc. Como podem existir bilhões desses endereços, manter o controle de posições de armazenamento individuais poderá vir a ser muito difícil. Para simplificar a armazenagem de informações, os programas definem *variáveis*, que são os nomes que o programa associa com posições específicas na memória. Como a palavra variável implica, o *valor* que o programa armazena nessas posições pode ser modificado durante a vida do programa.

Cada variável tem um *tipo* específico, que diz ao computador quanta memória requer os dados que a variável armazena, e quais operações o programa pode realizar com os dados. Dado o exemplo anterior de um programa que imprime um documento, o programa poderia usar uma variável chamada *nomearq* (que armazena o nome do arquivo que você deseja imprimir) e uma chamada *conta* (que armazena o número de cópias que você deseja imprimir). Dentro do seu programa C você referencia as variáveis pelo nome. Portanto, deve atribuir nomes representativos a cada variável. Dentro de seus programas, você normalmente declara suas variáveis imediatamente após *main*, antes dos comandos do programa, como mostrado aqui:

```
void main(void)
 {
   // As variáveis ficam aqui

   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

O programa a seguir mostra como você declararia três variáveis inteiras (variáveis que armazenam números de contagem, tais como 1, 2, 3 etc.)

```
void main(void)
 {
   int idade;       // A idade do usuário em anos
   int peso;        // O peso do usuário em Kg
   int altura;      // A altura em centímetros

   // Os outros comandos do programa ficariam aqui
 }
```

Cada variável tem um tipo que define a quantidade de memória que ela requer, bem como as operações que o programa pode realizar com os dados. Para declarar uma variável inteira, seus programas C usam o tipo *int*. Após você declarar uma variável (isto é, dizer ao programa o nome e o tipo da variável), poderá então atribuir um valor à variável (isto é, armazenar informações).

24 ATRIBUINDO UM VALOR A UMA VARIÁVEL

Uma variável é um nome que seu programa associa com uma posição de armazenamento na memória. Após você declarar uma variável dentro de seu programa, poderá atribuir um valor à variável usando o sinal de igual (chamado *operador de atribuição*). O programa a seguir declara três variáveis do tipo *int* e depois atribui um valor a cada variável:

```
void main(void)
 {
   int idade;       // A idade do usuário em anos
```

```
    int peso;        // O peso do usuário em Kg
    int altura;      // A altura em centímetros

    idade = 41;      // Atribui a idade do usuário
    peso = 80;       // Atribui o peso
    altura = 182;    // Atribui a altura

    // Outros comandos do programa
}
```

COMPREENDENDO OS TIPOS DE VARIÁVEIS 25

Quando você declara variáveis dentro de seus programas, precisa dizer ao compilador C o nome e o tipo da variável. O tipo define um conjunto de valores que a variável pode armazenar, bem como as operações que o programa pode realizar com os dados. C suporta quatro tipos básicos, cada um dos quais está listado na Tabela 25.

Tabela 25 Os quatro tipos básicos suportados por C

Nome do Tipo	Propósito
char	Armazena um único caractere, tal como uma letra de A até Z.
int	Armazena os números de contagem (chamados inteiros), tais como 1, 2, 3, bem como números negativos.
float	Armazena números de ponto flutuante de precisão simples (com um ponto decimal), tais como 3.14 ou -54.1343
double	Armazena um número de ponto flutuante de dupla precisão (que é mais exato que um número em ponto flutuante de precisão simples) Você usará *double* para números muito pequenos ou muito grandes.

Muitas das dicas apresentadas neste livro examinam cada um desses tipos detalhadamente. A maioria das dicas usa uma ou mais variáveis dos tipos básicos.

DECLARANDO MÚLTIPLAS VARIÁVEIS DO MESMO TIPO 26

Como você aprendeu na Dica 24, ao declarar uma variável dentro de seu programa, é preciso dizer ao compilador C o nome e o tipo da variável. Os comandos a seguir declaram três variáveis do tipo *int*:

```
int idade;
int peso;
int altura;
```

Quando você declarar variáveis do mesmo tipo, C lhe permite listar os nomes das variáveis em uma ou mais linhas, separando o nome de cada variável por vírgulas, como mostrado aqui:

```
int idade, peso, altura;
float salario, impostos;
```

COMENTANDO SUAS VARIÁVEIS NA DECLARAÇÃO 27

Os comentários ajudam qualquer um que analisar o programa a compreendê-lo mais facilmente. Quando você escolher nomes de variáveis, deverá selecionar nomes que descrevam claramente o valor que a variável armazenará. Por exemplo, considere a seguinte declaração:

```
int idade, peso, altura;
int x, y, z;
```

Ambas as declarações criam três variáveis do tipo *int*. No entanto, na primeira declaração, você tem uma idéia de como usar cada variável simplesmente examinando os nomes delas. Além de usar nomes representativos, você também deverá colocar um comentário ao lado de cada declaração de variável, o que explicará ainda melhor a variável, como mostrado aqui:

```
int idade;      // A idade do usuário em anos
int peso;       // O peso do usuário em Kg
int altura;     // A altura em centímetros
```

28 ATRIBUINDO VALORES ÀS VARIÁVEIS NA DECLARAÇÃO

Após você declarar uma variável dentro de seu programa, poderá usar o *operador de atribuição* (o sinal de igual) para atribuir um valor à variável. C lhe permite atribuir um valor a uma variável no momento em que ela é declarada. Os programadores chamam esse processo de atribuir um valor inicial de *inicialização* da variável. Os comandos a seguir, por exemplo, declaram e inicializam três variáveis do tipo *int:*

```
int idade = 41;      // A idade do usuário em anos
int peso = 80;       // O peso do usuário em Kg
int altura = 182;    // A altura em centímetros
```

29 INICIALIZANDO MÚLTIPLAS VARIÁVEIS DURANTE A DECLARAÇÃO

Na Dica 26, você aprendeu que C lhe permite declarar duas ou mais variáveis na mesma linha, como mostrado aqui:

```
int idade, peso, altura;
```

Quando você declara múltiplas variáveis na mesma linha, C lhe permite inicializar uma ou mais variáveis:

```
int idade = 44, peso, altura = 182;
```

Neste exemplo, C inicializará as variáveis *idade* e *altura,* e deixará a variável *peso* não inicializada.

30 USANDO NOMES REPRESENTATIVOS PARA AS VARIÁVEIS

Quando você declarar variáveis nos seus programas, deverá escolher nomes representativos que descrevam o uso da variável. Você poderá usar uma combinação de letras maiúsculas e minúsculas nos nomes de suas variáveis. Como discutido na Dica 8, o compilador C faz distinção entre letras maiúsculas e minúsculas. Se você usar letras maiúsculas e minúsculas nos nomes de suas variáveis, sempre precisará especificar as mesmas combinações de letras. Agora que você está começando, deve provavelmente adotar apenas letras minúsculas, pois fazer isso reduz a possibilidade de ocorrência de erros devido à mistura dos tipos de letras.

Você precisará dar um nome único a cada variável que declarar dentro de seus programas. Em geral, você pode usar um número ilimitado de caracteres no nome de uma variável. Os nomes das suas variáveis podem conter uma combinação de letras, números e o caractere de sublinhado, no entanto, os nomes devem iniciar com uma letra ou com o sublinhado. Os comandos a seguir ilustram alguns nomes válidos de variáveis:

```
int horas_trabalhadas;
float aliquota_imposto;
float _6_mes_gravidez;   // É válido iniciar com o _sublinhado
```

C predefine várias palavras-chave que têm significado especial para o compilador. Uma *palavra-chave* é uma palavra que tem significado para o compilador. Por exemplo, *float, int* e *char* são todas palavras-chave. Quando você criar nomes de variáveis, não poderá usar palavras-chave próprias. A Dica 31 lista as palavras-chave da linguagem C.

31 COMPREENDENDO AS PALAVRAS-CHAVE DE C

A linguagem C define várias palavras-chave que têm significado especial para o compilador. Quando você escolher nomes de variáveis (e criar suas próprias funções), não use essas palavras-chave. A Tabela 31 lista as palavras-chaves de C:

Tabela 31 Lista de palavras-chave de C.

Palavras-chave de C					
auto	*default*	*float*	*register*	*struct*	*volatile*
break	*do*	*for*	*return*	*switch*	*while*
case	*double*	*goto*	*short*	*typedef*	
char	*else*	*if*	*signed*	*union*	
const	*enum*	*int*	*sizeof*	*unsigned*	
continue	*extern*	*long*	*static*	*void*	

COMPREENDENDO AS VARIÁVEIS DO TIPO INT 32

Uma *variável* é um nome que o compilador C associa com uma ou mais posições de memória. Quando você declara uma variável dentro de seu programa, precisa especificar o tipo e o nome da variável. O *tipo* da variável especifica o tipo de valores que a variável pode armazenar e o conjunto de operações que o programa pode realizar com os dados. C usa o tipo *int* para armazenar valores inteiros (números de contagem positivos e negativos). O compilador C normalmente aloca dezesseis bits (dois bytes) para armazenar os valores do tipo *int*. Uma variável do tipo *int* pode armazenar valores no intervalo de -32.768 até 32.767. A Figura 32 mostra como C representa um valor inteiro

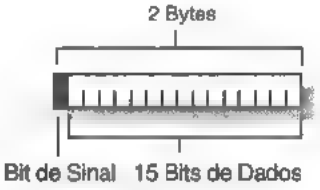

Figura 32 Como C representa um valor inteiro.

Os valores do tipo *int* são números inteiros; eles não incluem uma parte fracionária como os números em ponto flutuante. Se você atribuir um valor de ponto flutuante a uma variável do tipo *int*, a maioria dos compiladores C simplesmente truncará a parte fracionária. Se você atribuir uma variável do tipo *int* a um valor fora do intervalo de -32.768 até 32.767, uma condição de extravasamento ocorrerá, e o valor atribuído estará errado.

COMPREENDENDO AS VARIÁVEIS DO TIPO CHAR 33

Uma *variável* é um nome que o compilador C associa com uma ou mais posições de memória. Quando você declara uma variável dentro de seu programa, precisa especificar o nome e o tipo da variável. O *tipo* de uma variável especifica o tipo de valores que a variável pode armazenar e o conjunto de operações que o programa pode realizar com os dados. C usa o tipo *char* para armazenar valores de caracteres (bytes). O compilador C normalmente aloca oito bits (um byte) para armazenar os valores do tipo *char*. Uma variável do tipo *char* pode armazenar valores de números inteiros no intervalo de -128 a 127. A Figura 33 mostra como C representa um valor do tipo *char*.

Figura 33 Como C representa um valor do tipo *char*.

Os programas podem atribuir um valor a uma variável do tipo *char* de dois modos. Primeiro, o programa pode atribuir o valor ASCII de um caractere. Por exemplo, a letra A tem o valor ASCII 65:

```
char letra = 65; // Atribui o caractere A à variável letra
```

Segundo seu programa pode usar uma constante de caractere, que aparece dentro de apóstrofos, como mostrado aqui:

```
char letra = 'A';
```

As variáveis do tipo *char* somente contêm uma letra de cada vez. Para armazenar múltiplos caracteres, você precisa declarar uma string de caracteres, o que é discutido mais à frente na seção Strings.

34 COMPREENDENDO AS VARIÁVEIS DO TIPO FLOAT

Uma *variável* é um nome que o compilador C associa com uma ou mais posições de memória. Quando você declara uma variável dentro do seu programa, precisa especificar o tipo e o nome da variável. O *tipo* de uma variável especifica o tipo de valores que a variável pode armazenar e o conjunto de operações que o programa pode realizar com os dados. C usa o tipo *float* para armazenar valores em ponto flutuante (números positivos e negativos que contêm partes fracionárias). O compilador C normalmente aloca 32 bits (4 bytes) para armazenar valores do tipo *float*. Uma variável do tipo *float* pode armazenar valores com seis a sete dígitos de precisão no intervalo de 3.4E-38 até 3.4E+38.

C armazena o valor como uma *mantissa* de 23 bits, que contém o número fracionário; um expoente de 8 bits, que contém a potência a qual o computador elevará o número quando for determinar seu valor; e um bit de sinal, que determina se o valor é positivo ou negativo. Em outras palavras, se uma variável continha o valor 3.4E+38, o bit de sinal seria 0, o que indica que o número é positivo, a mantissa de 23 bits incluiria uma representação binária de 3.4, e o expoente de 8 bits incluiria uma representação binária do expoente 10^{38}. A Figura 34 ilustra como C representa um valor do tipo *float*. A Dica 337 explica as mantissas e os expoentes em detalhes.

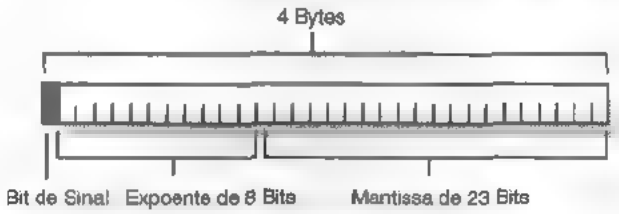

Figura 34 Como C representa um valor do tipo *float*

Nota: Esta dica e outras dicas posteriores representam os números em ponto flutuante em notação científica. Basicamente a notação científica permite-lhe representar qualquer número como um dígito simples à esquerda do ponto decimal, e um número ilimitado de dígitos à direita do ponto decimal, e um expoente representando 10 elevado ao valor desse expoente. Para determinar o valor real do número, multiplique o número (a ***mantissa****) pelo valor 10^{x} (onde* ***x*** *representa o* ***expoente****). Por exemplo, o número 3.1415967E+7 é avaliado como 31415967.0 ou 3.1415967 * 10^{7}.*

35 COMPREENDENDO AS VARIÁVEIS DO TIPO DOUBLE

Uma *variável* é um nome que o compilador C associa com uma ou mais posições de memória. Quando você declara uma variável dentro de seu programa, precisa especificar o tipo e o nome da variável. O *tipo* da variável especifica o tipo de valores que as variáveis podem armazenar e o conjunto de operações que o programa pode realizar com os dados. C usa o tipo *double* para armazenar valores em ponto flutuante (números positivos e negativos que contêm as partes fracionárias). O compilador C normalmente aloca 64 bits (8 bytes) para armazenar os valores do tipo *double*. Uma variável do tipo *double* pode armazenar valores com 14 a 15 dígitos de precisão, no intervalo de -1.7E-308 até 1.7E+308. A Figura 35 ilustra como C representa um valor do tipo *double*.

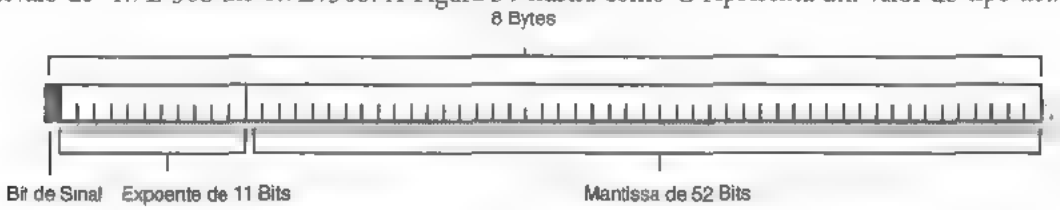

Figura 35 Como C representa um valor do tipo *double*

ATRIBUINDO VALORES AOS VALORES EM PONTO FLUTUANTE 36

Um valor em *ponto flutuante* é um valor que contém uma parte fracionária, tal como 123.45. Ao trabalhar com valores em ponto flutuante dentro de seus programas, você pode referenciar os valores usando seus formatos decimais, tais como 123.45, ou pode usar o formato exponencial do número, 1.2345E2. Portanto, os seguintes comandos atribuem o mesmo valor à variável *raio:*

```
raio = 123.45;
raio = 1.2345E2;
```

De forma similar, os dois comandos a seguir atribuem o mesmo valor fracionário à variável *raio.*

```
raio = 0.12345;
raio = 12.345E-2;
```

COMPREENDENDO OS MODIFICADORES DE TIPO 37

C oferece quatro tipos básicos de dados *(int, char, float e double).* Como você aprendeu, cada tipo define um conjunto de valores que a variável pode armazenar e um conjunto de operações que o programa pode realizar com os dados. Já vimos que as variáveis do tipo *int* podem armazenar valores no intervalo de -32.768 até 32 767. Da mesma forma, as variáveis do tipo *char* armazenam valores no intervalo de -128 até 127. Para lhe ajudar a modificar o intervalo de valores que as variáveis do tipo *int* e *char* podem armazenar, C oferece um conjunto de modificadores de tipo — *unsigned, long, register, signed* e *short.* Um *modificador de tipo* muda (modifica) o intervalo de valores que uma variável pode armazenar ou o modo como o compilador armazena uma variável. Para modificar um tipo, coloque o modificador de tipo diante do nome do tipo na declaração da variável, como mostrado aqui:

```
unsigned int numero_itens;
register int contador;
long int numero_muito_grande;
```

Várias outras dicas posteriores discutirão os quatro modificadores de tipo em detalhes.

COMPREENDENDO O MODIFICADOR DE TIPO UNSIGNED 38

Um *modificador de tipo* altera (modifica) o intervalo de valores que uma variável pode armazenar ou o modo como o compilador armazena uma variável. Como você aprendeu, as variáveis do tipo *int* podem armazenar valores positivos e negativos no intervalo de -32 768 até 32 767. Dentro da representação de um valor do tipo *int,* o bit mais significativo do valor indica o sinal do valor (positivo ou negativo), como vimos na Dica 32. Em alguns casos, seu programa pode nunca precisar armazenar um valor negativo em uma variável específica. O modificador de tipo *unsigned* diz ao compilador para não usar o bit mais significativo como um bit de sinal, mas, em vez disso, permitir que o bit represente valores positivos maiores. Uma variável do tipo *unsigned int* pode armazenar valores no intervalo de 0 até 65.535. A Figura 38.1 ilustra como o compilador C armazena uma variável do tipo *unsigned int.*

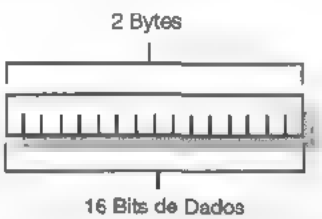

Figura 38.1 Como o compilador C representa os valores do tipo ***unsigned int.***

Como discutido na Dica 33, as variáveis do tipo *char* podem conter valores no intervalo de -128 até 127. Ao usar o modificador de tipo *unsigned* com as variáveis do tipo *char,* você pode criar variáveis que podem armazenar valores no intervalo de 0 até 255. A Figura 38.2 ilustra como o compilador C representa uma variável *unsigned char.*

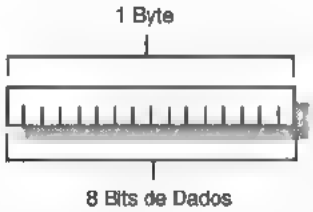

Figura 38.2 Como o compilador C representa as variáveis do tipo ***unsigned char***.

Os comandos a seguir ilustram as declarações das variáveis com tipo *unsigned int* ou *unsigned char*:

```
void main(void)
 {
  unsigned int segundo_atuais;
  unsigned int indicador_status;
  unsigned char borda_menu;  // Caractere ASCII estendido
 }
```

39 COMPREENDENDO O MODIFICADOR DE TIPO LONG

Um modificador de tipo altera (modifica) o intervalo de valores que uma variável pode armazenar ou o modo como o compilador armazena uma variável. Variáveis do tipo *int* podem armazenar valores positivo e negativo no intervalo -32.768 até 32.767. Como mostrado anteriormente na Dica 32, o compilador C representa os valores do tipo *int* usando 16 bits, com o bit mais significativo indicando o sinal do valor. Em muitos casos, seus programas podem armazenar valores inteiros que sao maiores (maiores que 32.767) ou menores (menores que -32.768) que o intervalo de valores que uma variável do tipo *int* pode conter. O modificador de tipo *long* diz ao compilador para usar 32 bits (4 bytes) para representar os valores inteiros. Uma variável do tipo *long int* pode armazenar valores no intervalo de -2.147.483.648 até 2.147.483.647. A Figura 39 mostra como o compilador C armazena uma variável *long int*.

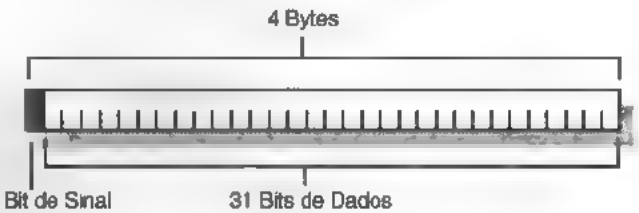

Figura 39 Como o compilador C representa os valores do tipo ***long int***. .

Nota : Muitos compiladores C++ também suportam o tipo ***long double****, que seus programas podem usar para representar os números em ponto flutuante com até 80 dígitos de precisão, em vez do padrão de 64 dígitos de precisão. Os valores do tipo long double usam 10 bytes de memória, com uma mantissa de 60 bits e um expoente de 19 bits. O intervalo para um valor long double é 3.4E-4932 até 1.1E+4932. Para determinar se seu compilador suporta declarações* ***long double****, verifique a documentação do seu compilador.*

40 COMBINANDO OS MODIFICADORES DE TIPO UNSIGNED E LONG

Na Dica 38 você aprendeu que o modificador de tipo *unsigned* instrui o compilador C a não interpretar o bit mais significativo de um valor como um indicador de sinal, mas, em vez disso, instrui a usar esse bit para representar um valor maior. Da mesma forma, na Dica 39, você aprendeu que o modificador de tipo *long* instrui o compilador a dobrar o número de bits que ele usa para representar um valor inteiro. Em alguns casos, seus programas podem precisar armazenar valores positivos muito grandes. Combinando os modificadores de tipo *unsigned* e *long*, você pode instruir o compilador C a alocar uma variável de 32 bits capaz de armazenar valores no intervalo de 0 até 4.292.967 265. A Figura 40 ilustra como o compilador C representaria uma variável *unsigned long int*:

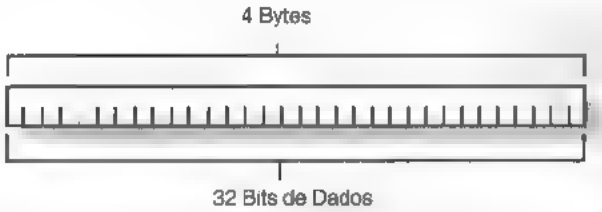

Figura 40 Como o compilador C representa valores do tipo *unsigned long int*.

Os comandos a seguir declaram variáveis do tipo *unsigned long int*:

```
void main(void)
 {
   unsigned long int valor_muito_grande;
   unsigned long int divida_publica;
 }
```

TRABALHANDO COM OS VALORES GRANDES 41

Como você aprendeu, as variáveis do tipo *int* podem armazenar valores no intervalo de -32.768 até 32.767. Da mesma forma, as variáveis do tipo *long int* podem armazenar valores no intervalo de -2.147.483.648 até 2.147.483.647. Quando você trabalhar com valores grandes em seus programas, não inclua vírgulas. Em vez disso, trabalhe com os números grandes como mostrado aqui:

```
long int numero_grande = 1234567;
long int um_milhao = 1000000;
```

Se você incluir vírgulas dentro de seus números, o compilador C gerará um erro de sintaxe.

COMPREENDENDO O MODIFICADOR DE TIPO REGISTER 42

Uma variável é o nome que seu programa associa com uma posição de memória. Quando você declara uma variável, o compilador C aloca memória para conter o valor da memória. Quando seu programa precisar acessar a variável, ocorre uma pequena sobrecarga na execução (o computador consome tempo) enquanto o processador acessa a memória. Dependendo do uso da variável, algumas vezes você pode instruir o compilador a armazenar a variável em um registrador (que reside dentro da própria CPU) para aumentar o desempenho do seu programa. Como o compilador pode acessar o valor muito mais rapidamente quando ele reside em um registrador, a execução do seu programa acaba sendo mais rápida. O modificador de tipo *register* instrui o compilador a manter a variável em um registrador sempre que for possível. Como o processador tem um número limitado de registradores, o compilador não pode manter o valor de uma variável permanentemente em um registrador. Assim, o compilador tentará manter a variável em um registrador sempre que for possível. Os comandos a seguir mostram o uso do modificador de tipo *register*:

```
void main(void)
 {
   register int contador;
   register unsigned sinaliz_status;
 }
```

Você deverá usar o modificador de tipo *register* com as variáveis que seu programa acessar repetidamente, tais como variáveis controladoras de um *laço* que o programa acessa a cada iteração.

COMPREENDENDO O MODIFICADOR DE TIPO SHORT 43

Como discutido na Dica 32, o compilador C normalmente representa as variáveis do tipo *int* usando 16 bits. Portanto, as variáveis do tipo *int* podem armazenar valores no intervalo de -32.768 até 32.767. No entanto, se você estiver usando um compilador de 32 bits, o compilador pode representar um valor inteiro usando 32 bits, o que significa que uma variável do tipo *int* poderia armazenar valores no intervalo de -2.147.483.648 até 2.147.483.647. Se você armazenar um valor que esteja fora do intervalo que uma variável do tipo *int* pode ar-

mazenar, ocorre uma condição de extravasamento, e o valor atribuído estará errado (A Dica 50 explica o extravasamento em detalhes.) Os programadores escrevem alguns programas sabendo que, ao ocorrer um extravasamento, o compilador consistentemente atribuirá o valor errante (o que significa que o valor errante é sempre o mesmo) ao valor extravasado. Em outras palavras, o programador escreve o programa para usar extravasamento. Se você migrar um programa que usa valores do tipo *int* desse modo (isto é, que conta com o extravasamento do valor) de um ambiente de 16 para 32 bits, o extravasamento não ocorrerá mais, pois o inteiro de 32 bits poderá armazenar um valor maior. Se você escrever um programa que se baseia em extravasamento, o que presume que o compilador representa as variáveis *int* com 16 bits, você poderá usar o modificador de tipo *short* para garantir que o compilador represente uma variável usando 16 bits. Os comandos a seguir ilustram as declarações de variáveis do tipo *short int:*

```
void main(void)
 {
   short int valor_chave;
   short int numero_pequeno;
 }
```

44 OMITINDO INT DAS DECLARAÇÕES MODIFICADAS

Dentro desta seção, você aprendeu sobre vários modificadores de tipo de C, incluindo *long, short* e *unsigned.* Os comandos a seguir ilustram como usar esses três modificadores.

```
unsigned int sinaliz_status;
short int valor_pequeno;
long int numero_muito_grande;
```

Quando você usar esses três modificadores de tipo, a maioria dos compiladores permitirá que você omita o *int,* como mostrado aqui:

```
unsigned sinaliz_status;
short valor_pequeno;
long numero_muito_grande;
```

45 COMPREENDENDO O MODIFICADOR DE TIPO SIGNED

Como você aprendeu na Dica 33, os compiladores C normalmente representam as variáveis do tipo *char* usando oito bits, com o bit mais significativo representando o sinal do valor. Portanto, as variáveis do tipo *char* podem armazenar valores no intervalo de -128 até 127. Na Dica 38, você aprendeu que pode usar o qualificador *unsigned* para instruir o compilador C a não interpretar o bit de sinal, mas, em vez disso, a usar o bit para representar um valor positivo maior. Usando o modificador de tipo *unsigned,* uma variável do tipo *char* pode armazenar valores no intervalo de 0 até 255. Se você estiver usando uma variável do tipo *char* e atribuir à variável um valor fora do intervalo de valores válidos, haverá extravasamento e o valor que o computador atribuirá à variável não será aquele que você deseja. No entanto, em alguns casos, você escreverá programas planejando a ocorrência de extravasamento. Se você acha que futuramente esse programa será compilado sob um compilador diferente, que talvez represente as variáveis do tipo *char* como unsigned, é possível usar o modificador de tipo *signed* para garantir que o segundo compilador represente as variáveis do tipo *char* usando 7 bits para os dados e 1 bit para o sinal. Os comandos a seguir mostram declarações do tipo *signed char:*

```
void main(void)
 {
   signed char valor_byte;
   signed char escolha_menu;
 }
```

46 MÚLTIPLAS OPERAÇÕES DE ATRIBUIÇÃO

Como você aprendeu, C usa o sinal de igual (=) como um operador de atribuição. Normalmente, seus programas C atribuirão valores às variáveis em linhas distintas, como mostrado aqui:

```
conta = 0;
soma = 0;
valor = 0;
```

Quando você quiser atribuir o mesmo valor a múltiplas variáveis, C lhe permite efetuar todas as atribuições de uma só vez, como mostrado aqui:

```
conta = soma = valor = 0;
```

Quando C encontra uma operação de atribuição múltipla, atribui os valores da direita para a esquerda. Como regra, somente use atribuições múltiplas para inicializar variáveis. Usar essas operações para cálculos mais complexos prejudica a legibilidade do seu programa. Por exemplo, o programa a seguir atribuirá a duas variáveis o equivalente maiúsculo do caractere digitado pelo usuário:

```
salva_letra = letra = toupper(getchar());
```

ATRIBUINDO O VALOR DE UM TIPO DE VARIÁVEL A UM TIPO DIFERENTE 47

Um tipo define o conjunto de valores que uma variável pode armazenar e o conjunto de operações que seus programas podem executar com os dados. C oferece quatro tipos básicos de dados (*int, float, char* e *double*). Alguns casos podem requerer que você atribua o valor de uma variável do tipo *int* a uma variável do tipo *float*, e vice-versa. Como uma regra geral, você pode atribuir com sucesso um valor do tipo *int* a uma variável do tipo *float*. No entanto, quando atribuir o valor de uma variável do tipo *float* a uma variável do tipo *int*, precisa tomar cuidado. A maioria dos compiladores truncará o valor em ponto flutuante, descartando a parte fracionária. Porém, outro compilador poderia arredondar o valor em vez de simplesmente truncá-lo (isto é, se a parte fracionária for maior que 0.5, os dois compiladores converterão o valor de forma diferente). Se você quiser garantir que seu programa efetue a conversão de valores em ponto flutuante para inteiro de forma consistente, considere o uso das funções *ceil* e *floor*, apresentadas mais à frente na seção Matemática.

CRIANDO SEUS PRÓPRIOS TIPOS 48

Um tipo define o conjunto de valores que uma variável pode armazenar e o conjunto de operações que seu programa pode realizar com os dados. C oferece quatro tipos básicos de dados (*int, float, char* e *double*). Como visto você pode combinar os modificadores de tipo para alterar o intervalo de valores que uma variável pode armazenar. À medida que o número de variáveis declaradas aumentar, você poderá achar conveniente criar seu próprio nome de variável, o que oferece um nome de atalho para um tipo usado comumente. Por exemplo, considere as seguintes declarações do tipo *unsigned long int*:

```
unsigned long int segundos_desde_janeiro;
unsigned long int populacao_mundial_em_2000;
```

Usando o comando *typedef* de C, você pode definir o nome de tipo *ULINT*, que é idêntico ao tipo *unsigned long int*, como mostrado aqui:

```
typedef unsigned long int ULINT;
```

Após criar o nome do tipo, você poderá usar o nome para definir variáveis, como mostrado aqui:

```
ULINT segundos_desde_janeiro;
ULINT populacao_mundial_em_2000;
```

À medida que seus programas forem usando declarações de variáveis mais complexas, você verá que criar um novo nome de tipo é muito conveniente, pois reduz a quantidade de digitação e a possibilidade de erros.

Nota: O código dentro desta dica define ULINT em maiúsculas porque é mais fácil para outro programador determinar os tipos personalizados se você os representar de forma diferente que os tipos padrão. Você pode usar letras maiúsculas, minúsculas ou uma combinação das duas nos nomes dos tipos - a escolha é sua. No entanto, seja coerente ao nomear os tipos personalizados em seus programas ou os diversos tipos dentro do mesmo programa.

49 ATRIBUINDO UM VALOR HEXADECIMAL OU OCTAL

Dependendo do seu aplicativo, algumas vezes você precisará trabalhar com valores *octais* (base 8) ou *hexadecimais* (base 16). Nessas ocasioes você dirá ao compilador que quer trabalhar com valores que nao são decimais. Se você preceder um valor numérico com um 0 (zero), tal como 077, o compilador C tratará o valor como octal. Da mesma forma, se você preceder um valor com 0x, tal como 0xFF, o compilador tratará o valor como hexadecimal. Os comandos a seguir ilustram como usar uma constante octal e hexadecimal.

```
int valor_octal = 0227;
int valor_hexa = 0xFF0;
```

50 COMPREENDENDO O EXTRAVASAMENTO

Como você aprendeu, o tipo de uma variável define o intervalo de valores que uma variável pode armazenar e as operações que um programa pode realizar com a variável. Por exemplo, as variáveis do tipo *int* podem armazenar valores no intervalo de -32.768 até 32.767. Se você atribuir um valor fora desse intervalo a uma variável do tipo *int*, ocorrerá um erro de extravasamento. Como você já aprendeu, C usa 16 bits para representar as variáveis do tipo *int*. O compilador C usa o mais significativo dos 16 bits para determinar o sinal de uma variável. Se o bit mais significativo for 0, o valor será positivo. Se o bit mais significativo for 1, o valor será negativo. C depois usa 15 bits para representar o valor da variável. Para compreender por que ocorre o extravasamento, você precisa considerar a implementação bit a bit do valor. Considere os seguintes valores:

```
     0    0000 0000 0000 0000
     1    0000 0000 0000 0001
     2    0000 0000 0000 0010
     3    0000 0000 0000 0011
     4    0000 0000 0000 0100

32.765    0111 1111 1111 1101
32.766    0111 1111 1111 1110
32.767    0111 1111 1111 1111
```

Se você somasse 1 ao valor seria 32.767, poderia esperar o resultado 32.768. No entanto, para C o valor seria -32.768, como mostrado aqui:

```
 32.767    0111 1111 1111 1111
 +    1    0000 0000 0000 0001
-------    -------------------
-32.768    1000 0000 0000 0000
```

O programa a seguir, *extravas.c*, ilustra como ocorre o extravasamento:

```
#include <stdio.h>

void main(void)
 {
   int positivo = 32767;
   int negativo = -32768;

   printf("%d + 1 é %d\n", positivo, positivo+1);
   printf("%d - 1 é %d\n", negativo, negativo-1);
 }
```

Quando você compilar e executar este programa, sua tela exibirá a seguinte saída:

```
32767 + 1 é -32768
-32768 -1 é 32767
C:\>
```

Como você pode ver, somar um valor a 32.767 produz um número negativo, enquanto subtrair um valor de -32.768 produz um número positivo. Um problema que dificulta o extravasamento é que, dentro de seus pro-

gramas, você normalmente não verá o erro porque o compilador C não retorna um erro quando o extravasamento ocorre Em outras palavras, a execução do programa continua, apesar do extravasamento. Consequentemente quando você depurar seus programas, poderá ter dificuldade em detectar os erros que resultam de um extravasamento.

Nota: Se você usar o compilador Turbo C++ Lite, ou a maioria dos compiladores mais novos (tais como o Microsoft Visual C++, o Borland C++ 5.02®), o compilador o advertirá sobre um problema potencial de extravasamento O compilador Turbo C++ Lite fará advertências de ***Constant is long in function main*** *e* ***Conversion may lose significant digits in function main()****, mas, mesmo assim, executará o programa (e extravasará as variáveis). Como regra geral, mesmo se uma mensagem de advertência do compilador não interromper a compilação do programa, você deverá observar essas mensagens com atenção e agir apropriadamente*

COMPREENDENDO A PRECISÃO 51

Como vimos, os computadores representam os números internamente usando combinações de 1s e 0s (dígitos binários). Nas dicas anteriores, você aprendeu que, como um tipo tem um numero fixo de bits, cada tipo somente pode conter um intervalo específico de valores. Se você atribuiu um valor fora do intervalo do tipo, ocorrerá um erro de extravasamento. Os números em ponto flutuante podem experimentar extravasamento e ter precisão insuficiente. A *precisão* de um valor define seu grau de exatidão. Os valores do tipo *float*, por exemplo, oferecem de seis a sete dígitos significativos. Por exemplo, assuma que você atribuiu o valor 1.234567890 a uma variável do tipo *float*. Como o tipo *float* somente oferece sete dígitos significativos, você somente pode contar com precisão até 1.23456. Por outro lado, os valores do tipo *double*, oferecem 14 a 15 dígitos significativos Como resultado, um valor do tipo *double* poderia armazenar com exatidão o valor 1.234567890

Quando você trabalhar com numeros em ponto flutuante, você precisa saber que o computador representa os valores usando um número fixo de bits. Portanto, é impossível para o computador sempre representar os valores com exatidão Por exemplo, o computador pode representar o valor 0.4 como 0.3999999, ou o valor 0.1 como 0.099999, e assim por diante. O programa a seguir, *preciso.c*, ilustra a diferença entre precisão dupla e simples:

```
#include <stdio.h>

void main(void)
 {
   float exato = 0.123456790987654321;
   double mais_exato = 0.123456790987654321;

   printf("Valor de float\t %21.19f\n", exato);
   printf("Valor de double\t %21.19f\n", mais exato);
 }
```

Quando você compilar e executar o programa *preciso c*, sua tela exibirá o seguinte resultado:

```
Valor de float:  0.1234567890432815550
Valor de double: 0.1234567890987654380
C:\>
```

ATRIBUINDO APÓSTROFOS E OUTROS CARACTERES 52

À medida que você trabalhar com variáveis do tipo *char*, ou com strings de caracteres, algumas vezes precisará atribuir um caractere de apóstrofo ou de aspas a uma variável. Por exemplo, para escrever *Caixa d'agua*, você precisa incluir o apóstrofo dentro da string. Nesses casos, você precisa colocar o caractere dentro de apóstrofos precedido por uma barra invertida (\), como mostrado aqui:

```
char apostrofe = '\'';
char aspas = '\"';
```

Além dos caracteres de apóstrofo, seus programas podem requerer que você atribua um dos caracteres especiais listados na Tabela 52. Para fazer isso, simplesmente coloque o símbolo do caractere imediatamente após o caractere de barra invertida. Em todos os casos, você precisa usar letras minúsculas para representar o caractere especial.

Tabela 52 Caracteres de escape definidos por C.

Caractere de Escape	Significado
\a	Caractere ASCII do aviso sonoro
\b	Caractere de retrocesso
\f	Caractere de avanço do formulário
\n	Caractere de nova linha
\r	Retorno do carro (sem alimentação de linha)
\t	Tabulação horizontal
\v	Tabulação vertical
\\	Caractere da barra invertida
\'	Apóstrofo
\"	Aspas
\?	Interrogação
\nnn	Valor ASCII em octal
\xnnn	Valor ASCII em hexadecimal

53 INTRODUÇÃO AO COMANDO PRINTF

Várias dicas apresentadas neste livro usam a função *printf* para exibir mensagens na tela. Quando seu programa usa *printf*, as informações que você instrui *printf* a imprimir são chamadas *parâmetros* ou *argumentos de printf*. O comando a seguir usa *printf* para exibir a mensagem *Bíblia do Programador C/C++, do Jamsa!* na sua tela:

```
printf("Bíblia do Programador C/C++, do Jamsa!");
```

Neste caso, a string de caracteres (as letras que aparecem dentro das aspas) é o único parâmetro de *printf*. Quando seus programas começarem a trabalhar com variáveis, você poderá querer usar *printf* para exibir os valores de cada variável. A função *printf* suporta mais de um parâmetro. O primeiro parâmetro sempre precisa ser uma string de caracteres. Você pode fazer os parâmetros que seguem a primeira string de caracteres serem números, variáveis, expressões (tais como 3*15), ou até outras strings de caracteres. Quando você quiser que *printf* exiba um valor ou uma variável, precisa fornecer informações sobre o tipo da variável dentro do primeiro parâmetro. Além de especificar caracteres dentro do primeiro parâmetro, você pode incluir especificadores de formato, que instruem *printf* como imprimir os outros parâmetros. Esses especificadores de formato têm a forma de um sinal de porcentagem (%) seguido por uma letra. Por exemplo, para exibir um valor inteiro, você usa o *%d* (*d* para valor decimal). Da mesma forma, para imprimir um valor em ponto flutuante, você pode usar *%f*. Os seguintes comandos *printf* ilustram como você pode usar especificadores de formato com *printf*:

```
printf("A idade do usuário é %d\n", idade);
printf("O imposto é %f\n", custo * 0.07);
printf("Idade do usuário: %d  peso: %d  altura: %d\n,"
    idade, peso, altura);
```

Como você pode ver, dentro do primeiro parâmetro de *printf*, é possível especificar um ou mais especificadores de formato. Observe que o terceiro comando não coube na linha e continuou na linha seguinte. Quando seus comandos não couberem em uma linha, procure encontrar uma posição para quebrar a linha (tal como imediatamente após uma vírgula) e depois recue (endente) a linha seguinte. O propósito da endentação é melhorar o aspecto visual do seu programa e tornar evidente a qualquer pessoa que for ler seu programa que a linha é continuação da anterior. Várias dicas a seguir discutem em detalhes os diferentes especificadores de formato de *printf*.

54 EXIBINDO VALORES DO TIPO INT USANDO PRINTF

A função *printf* suporta especificadores de formato que oferecem a *printf* informações sobre seus tipos de parâmetros (tais como *int*, *float*, *char* e assim por diante). Para exibir valores do tipo *int* com *printf*, use o especificador de formato *%d*. O programa a seguir, *intsai.c* usa o especificador de formato *%d* para exibir valores e variáveis do tipo *int*:

```
#include <stdio.h>

void main(void)
```

```
{
  int idade = 41;
  int altura = 182;
  int peso = 80;

  printf("Idade do usuário: %d  peso: %d  altura: %d\n",
      idade, peso, altura);
  printf("%d mais %d igual a %d\n", 1, 2, 1 + 2);
}
```

Quando você compilar e executar o programa *intsai.c*, sua tela exibirá a seguinte saída.

```
Idade do usuário: 41  peso 80  altura 182
1 mais 2 igual a 3
C:\>
```

Nota: Muitos compiladores C tratam o especificador de formato %i como idêntico a %d. No entanto, se você estiver criando novos programas, use o especificador %d, pois %i é um legado do passado, e os compiladores futuros talvez deixem de aceitá-lo.

IMPRIMINDO UM VALOR INTEIRO OCTAL OU HEXADECIMAL 55

A função *printf* suporta especificadores de formato que oferecem a *printf* informações sobre seus tipos de parâmetros (tais como *int, float, char* e assim por diante). Dependendo do seu programa, algumas vezes você pode querer exibir um valor inteiro em seu formato octal (base 8) ou hexadecimal (base 16). O especificador de formato *%o* (letra o, não zero) instrui *printf* a exibir um valor em octal. De uma forma similar, os especificadores *%x* e *%X* instruem *printf* a exibir um valor no formato hexadecimal. A diferença entre *%x* e *%X* é que o último exibe valores hexadecimais em maiúsculas. O programa a seguir, *oct_hexa.c*, ilustra o uso dos especificadores de formato *%o, %x* e *%X:*

```
#include <stdio.h>

void main(void)
 {
   int valor = 255;

   printf("O valor decimal %d em octal é %o\n", valor, valor);
   printf("O valor decimal %d em hexadecimal é %x\n", valor, valor);
   printf("O valor decimal %d em hexadecimal é %X\n", valor, valor);
 }
```

Quando você compilar e executar o programa *oct_hexa.c*, sua tela exibirá a seguinte saída:

```
O valor decimal 255 em octal é 377
O valor decimal 255 em hexadecimal é ff
O valor decimal 255 em hexadecimal é FF
C:\>
```

EXIBINDO VALORES DO TIPO UNSIGNED INT USANDO PRINTF 56

Como você aprendeu, a função *printf* suporta especificadores de formato que fornecem a *printf* informações sobre seus tipos de parâmetro (tais como *int, char* etc.). Para exibir valores do tipo *unsigned int* com *printf*, você deve usar o especificador de formato *%u*. Se você usar *%d* em lugar de *%u*, *printf* tratará o valor especificado como tipo *int*, o que provavelmente exibirá o resultado incorreto. O programa seguinte *u_intsai.c*, usa o especificador de formato *%u*, bem como *%d*, para exibir o valor 42000. O programa *u_intsai.c* ilustra o tipo de erro que pode ocorrer se você usar o especificador de formato incorreto:

```
#include <stdio.h>

void main(void)
 {
   unsigned int valor = 42000;

   printf("Exibindo 42000 como unsigned %u\n", valor);
   printf("Exibindo 42000 como int %d\n", valor);
 }
```

Quando você compilar e executar o programa *u_intsai.c*, sua tela exibirá a seguinte saída:

```
Exibindo 42000 como unsigned 42000
Exibindo 42000 como int -23536
C:\>
```

Nota. Quando você compilar este programa sob o Turbo C++ Lite, o compilador exibirá duas mensagens de erro, pois ele vê o valor constante 42 000 que o programa tenta atribuir à variável ***unsigned int valor*** *como um número* ***long****, em vez de um* ***int****. Neste caso, como o propósito do programa é mostrar os erros que podem surgir a partir das declarações* ***unsigned int****, você deverá ignorar as advertências do compilador. Outros compiladores de 16 bits emitirão advertências similares.*

57 EXIBINDO VALORES DO TIPO LONG INT USANDO PRINTF

Como já vimos, a função *printf* suporta os especificadores de formato que lhe oferecem informações sobre os tipos de parâmetros (tais como *int, float, char* etc.). Para exibir valores do tipo *long int* com *printf*, você deverá usar o especificador de formato *%ld*. Se você usar *%d* em lugar de *%ld*, *printf* tratará o valor especificado como tipo *int*, muito provavelmente exibindo o resultado errado. O programa a seguir, *longsai.c*, usa o especificador de formato *%ld*, bem como *%d*, para exibir o valor 1000000. O programa *longsai.c* ilustra o tipo de erro que poderá ocorrer se você usar o especificador de formato incorreto:

```
#include <stdio.h>

void main(void)
 {
   long int um_milhao = 1000000;

   printf ('Um milhão é %ld\n", um_milhao);
   printf ('Um milhão é %d\n", um_milhao);

 }
```

Quando você compilar e executar o programa *longsai.c*, sua tela exibirá a seguinte saída:

```
Um milhão é 1000000
Um milhão é 16960
C:\>
```

58 EXIBINDO VALORES DO TIPO FLOAT USANDO PRINTF

Como você sabe, a função *printf* suporta especificadores de formato que lhe oferecem informações sobre os tipos de seus parâmetros (tais como *int, float, char* etc.). Para exibir valores do tipo *float* com *printf*, você deverá usar o especificador de formato *%f*. O programa a seguir, *floatsai.c*, usa o especificador de formato *%f* para exibir valores em ponto flutuante:

```
#include <stdio.h>

void main(void)
 {
   float preco = 525.75;
   float imposto_vendas = 0.06;
```

```
    printf("O custo do item é %f\n", preco);
    printf("O imposto sobre a venda do item é %f\n",
        preco * imposto_vendas);
}
```

Quando você compilar e executar o programa *floatsai.c*, sua tela exibirá a seguinte saída:

```
O custo do item é 525.750000
O imposto sobre a venda do item é 31.544999
C:\>
```

Como você pode ver, por padrão, o especificador de formato *%f* fornece pouca formatação na saída. No entanto, várias dicas nesta seção apresentam modos de formatar saída usando *printf*.

EXIBINDO VALORES DO TIPO CHAR USANDO PRINTF 59

Como você aprendeu, a função *printf* suporta especificadores de formato que fornecem informações sobre os tipos dos parâmetros (tais como *int, float, char* etc.). Para exibir os valores do tipo *char* com *printf*, você deverá usar o especificador de formato *%c*. O programa a seguir, *char_sai.c*, usa o especificador de formato *%c* para exibir a letra A na sua tela:

```
#include <stdio.h>

void main(void)
 {
   printf("A letra é %c\n", 'A');
   printf("A letra é %c\n", 65);
 }
```

Como você pode ver, o programa *char_sai.c* exibirá a letra A usando a constante de caractere 'A', bem como o valor ASCII 65. Quando você compilar e executar o programa *char_sai.c*, sua tela exibira a seguinte saída:

```
A letra é A
A letra é A
C:\>
```

EXIBINDO VALORES DE PONTO FLUTUANTE EM UM FORMATO EXPONENCIAL 60

Já vimos que, a função *printf* suporta os especificadores de formato que fornecem informações sobre os tipos de parâmetro (tais como *int, float, char*, etc.). Na Dica 58, você aprendeu que, usando o especificador de formato *%f*, é possível exibir valores em ponto flutuante. Dependendo dos requisitos do seu programa, algumas vezes você pode querer exibir os valores usando um formato exponencial. Para exibir um valor em ponto flutuante em um formato exponencial, use o especificador de formato *%e* ou *%E*. A diferença entre *%e* e *%E* é que este último instrui *printf* a usar uma letra E maiúscula na saída. O programa a seguir, *exp_sai.c*, usa os especificadores de formato *%e* e *%E* para exibir valores em ponto flutuante em seus formatos exponenciais:

```
#include <stdio.h>

void main(void)
 {
   float pi = 3.14159;
   float raio = 2.0031;

   printf("A área do círculo é %e\n", 2 * pi * raio);
   printf("A área do círculo é %E\n", 2 * pi * raio);
 }
```

Quando você compilar e executar o programa *exp_sai.c*, sua tela exibirá a seguinte saída:

```
A área do círculo é 1.258584e+01
```

```
A área do círculo é 1.258584E+01
C:\>
```

Como você pode ver, os especificadores de formato %e e %E oferecem pouca formatação da saída. No entanto, várias Dicas nesta seção apresentam modos de formatar a saída usando *printf*.

61 EXIBINDO VALORES EM PONTO FLUTUANTE

Na Dica 58 você aprendeu que, usando o especificador de formato *%f*, é possível instruir *printf* a exibir valores em ponto flutuante usando seus formatos em ponto decimal. Da mesma forma, na Dica 60, vimos que é possível usar os especificadores de formato *%e* e *%E* para instruir *printf* a exibir um valor em ponto flutuante usando um formato exponencial. De uma forma similar, *printf* suporta os especificadores de formato *%g* e *%G*. Quando você usa esses especificadores de formato, *printf* decide se deve usar o formato *%f* ou *%e*, dependendo da técnica que exibirá a saída no formato mais significativo para o usuário. O programa a seguir, *flt_pt.c*, ilustra o uso do especificador de formato *%g*.

```
#include <stdio.h>

void main(void)
 {
   printf("Exibir 0.1234 resulta em %g\n", 0.1234);
   printf("Exibir 0.00001234 resulta em %g\n", 0.00001234);
 }
```

Quando você compilar e executar o programa *flt_pt.c*, sua tela exibirá a seguinte saída:

```
Exibir 0.1234 resulta em 0.1234
Exibir 0.00001234 resulta em 1.234e-05
C:\>
```

62 EXIBINDO UMA STRING DE CARACTERES USANDO PRINTF

Uma string de caracteres é uma sequência de zero ou mais caracteres. (A seção Compreendendo as Strings, mais à frente, discute as strings de caracteres em maiores detalhes.) Uma das operações mais comuns que seus programas executarão é a saída de string de caracteres. Como você aprendeu, a função *printf* suporta os especificadores de formato que oferecem informações sobre seus tipos de parâmetro (tais como *int*, *float*, *char* etc.). Para exibir uma string de caracteres usando *printf*, você deverá usar o especificador de formato *%s*. O programa a seguir, *str_sai.c*, usa o especificador de formato *%s* para exibir uma string de caracteres:

```
#include <stdio.h>

void main(void)
 {
   char titulo[255] = "Bíblia do Programador C/C++, do Jamsa!";

   printf("O nome deste livro é %s\n", titulo);
 }
```

Quando você compilar e executar o programa *str_sai.c*, sua tela exibirá a seguinte saída:

```
O nome deste livro é Bíblia do Programador C/C++, do Jamsa!
C:\>
```

63 EXIBINDO UM ENDEREÇO DE PONTEIRO USANDO PRINTF

Como já foi visto, a função *printf* suporta especificadores de formato que informam os tipos dos parâmetros (tais como *int*, *float*, *char* etc.). Vimos também que uma variável é um nome que seu programa associa com uma posição de memória. À medida que a complexidade do seu programa aumentar, mais cedo ou mais tarde você trabalhará com os endereços de memória (chamados *ponteiros*). Quando você começar a trabalhar com ponteiros, algumas vezes precisará exibir o endereço de um ponteiro. Para exibir um endereço de ponteiro usando *printf*,

use o especificador de formato *%p*. O programa a seguir, *ptr_sai.c*, usa o especificador *%p* para exibir um endereço de memória:

```
#include <stdio.h>

void main(void)
 {
   int valor;

   printf("O endereço da variável valor é %p\n", &valor);
 }
```

Quando você compilar e executar o programa *ptr_sai.c*, sua tela exibirá a seguinte saída:

```
O endereço da variável valor é FFF4
C:\>
```

Quando você usar o especificador de formato *%p*, o valor atual do ponteiro e o formato que *printf* usa para exibir o valor diferirá de um sistema operacional para outro. A seção Ponteiros, mais à frente, discute em detalhes o uso de ponteiros.

Precedendo um Valor com um Sinal de Adição ou de Subtração 64

Como você aprendeu, *printf* suporta vários especificadores de formato que controlam como *printf* exibe a saída. Por padrão, quando você usa *printf* para exibir um valor negativo, ele precederá o valor com um sinal de subtração. Dependendo do seu programa, algumas vezes você pode querer que *printf* exiba o sinal para os valores positivos também. Para instruir *printf* a exibir o sinal de um valor simplesmente inclua um sinal de adição imediatamente após o % no especificador de formato. O programa a seguir, *exibesin.c*, ilustra o uso do sinal de adição dentro do especificador de formato:

```
#include <stdio.h>

void main(void)
 {
   int neg_int = -5;
   int pos_int = 5;
   float neg_flt = -100.23;
   float pos_flt = 100.23;

   printf("Os valores inteiros são %+d and %+d\n", neg_int, pos_int);
   printf("Os valores em ponto flutuante são %+f %+f\n", neg_flt, pos_flt);
 }
```

Quando você compilar e executar o programa *exibesin.c*, sua tela exibirá o seguinte:

```
Os valores inteiros são -5 e +5
Os valores em ponto flutuante são -100.230003 +100.230003
C:\>
```

Formatando um Valor Inteiro Usando printf 65

Como você leu na Dica 54, o especificador de formato *%d* instrui *printf* a exibir um valor inteiro. À medida que seus programas se tornarem mais complexos, você quererá que *printf* formate melhor seus dados. Por exemplo, assuma que você queira imprimir uma tabela na tela do seu computador que é similar à seguinte saída:

```
Vendedor   Quantidade
Jonas      332
Silva      1200
Alex       3311
Davi       43
```

Quando você usar o especificador de formato *%d*, poderá instruir *printf* a exibir um número mínimo de caracteres. O programa a seguir, *int_fmt.c*, ilustra como você poderia formatar valores inteiros usando *%d*:

```
#include <stdio.h>

void main(void)
  {
   int valor = 5;

   printf ("%1d\n", valor);
   printf ("%2d\n", valor);
   printf ("%3d\n", valor);
   printf ("%4d\n", valor);
  }
```

Quando você compilar e executar o programa *int_fmt.c*, sua tela exibirá o seguinte:

```
5
 5
  5
   5
C:\>
```

O dígito que você coloca após o % especifica o número mínimo de caracteres que *printf* usará para exibir um valor inteiro. Por exemplo, caso especifique *%5d* e o valor que você quer exibir for 10, *printf* precederá o valor com três espaços. Observe que o valor especifica o número mínimo de caracteres que a saída consumirá. Se o valor que você quer exibir requer mais caracteres do que você especificou, *printf* usará o número de caracteres que *printf* requer para exibir o valor corretamente.

66 SAÍDA DE INTEIROS PREENCHIDA COM ZEROS

Na Dica 65 você viu como formatar um valor inteiro colocando o número desejado de dígitos imediatamente após o % no especificador de formato *%d*. Se o valor inteiro que *printf* exibir não precisar do número de caracteres que você especificou, *printf* precederá o valor com o número de espaços necessários. Dependendo do propósito do seu programa, algumas vezes você pode querer que *printf* preceda o valor com zeros (chamado *zero de preenchimento*), em vez de espaços. Para instruir *printf* a preencher um valor com zeros, coloque um 0 (zero) imediatamente após o % no especificador de formato, antes do número desejado de dígitos. O programa a seguir, *enche.c*, ilustra o preenchimento com zeros:

```
#include <stdio.h>

void main(void)
  {
    int valor = 5;
   printf ("%01d\n", valor);
   printf ("%02d\n", valor);
   printf ("%03d\n", valor);
   printf ("%04d\n", valor);
  }
```

Quando você compilar e executar o programa *enche.c*, sua tela exibirá o seguinte:

```
5
05
005
0005
C:\>
```

EXIBINDO UM PREFIXO ANTES DOS VALORES OCTAIS OU HEXADECIMAIS 67

Na Dica 55 você aprendeu como usar o especificador de formato *%o* para exibir valores octais, e os especificadores de formato *%x* e *%X* para exibir valores hexadecimais. Quando seus programas apresentam esses valores, algumas vezes você pode querer preceder os valores octais com um zero (por exemplo, *0777*), e os hexadecimais com *0x* (por exemplo, *0xFF*).

Para instruir *printf* a preceder um valor octal ou hexadecimal com o prefixo apropriado, coloque um sinal de cerquinha (#) imediatamente após o % no especificador de formato. O programa a seguir, *exibe_oh.c*, ilustra o uso do caractere # no especificador de formato de *printf*:

```
#include <stdio.h>

void main(void)
 {
   int valor = 255;

   printf("O valor decimal %d em octal é %#o\n", valor, valor);
   printf("O valor decimal %d em hexadecimal é %#x\n", valor, valor);
   printf("O valor decimal %d em hexadecimal é %#X\n", valor, valor);
 }
```

Quando você compilar e executar o programa *exibe_oh.c*, sua tela exibirá o seguinte:

```
O valor decimal 255 em octal é 0377
O valor decimal 255 em hexadecimal é 0xff
O valor decimal 255 em hexadecimal é 0xFF
C:\>
```

FORMATANDO UM VALOR EM PONTO FLUTUANTE USANDO PRINTF 68

Na Dica 65 você aprendeu a formatar um valor inteiro colocando o número desejado de dígitos imediatamente após o % no especificador de formato *%d*. Usando uma técnica similar, *printf* lhe permite formatar a saída em ponto flutuante. Quando você formata um valor em ponto flutuante, especifica dois valores. O primeiro valor diz a *printf* o número mínimo de caracteres que você quer exibir. O segundo valor diz a *printf* o número de dígitos que você quer que sejam exibidos à direita do ponto decimal. O programa a seguir, *flt_fmt.c*, ilustra como formatar valores em ponto flutuante usando *printf*:

```
#include <stdio.h>

void main(void)
 {
   float valor = 1.23456;

   printf ("%8.1f\n", valor);
   printf ("%8.3f\n", valor);
   printf ("%8.5f\n", valor);
 }
```

Quando você compilar e executar o programa *flt_fmt.c*, sua tela exibirá o seguinte:

```
     1.2
   1.235
 1.23456
C:\>
```

FORMATANDO A SAÍDA EXPONENCIAL 69

Na Dica 68 foi visto como usar o especificador de formato *%f* para formatar valores em ponto flutuante. Usando técnicas de formatação similares, você pode instruir *printf* a exibir saída em ponto flutuante em um formato exponencial. O programa a seguir, *exp_fmt.c*, ilustra a saída formatada em exponencial.

```
#include <stdio.h>

void main(void)
 {
   float valor = 1.23456;

   printf ("%12.1e\n", valor);
   printf ("%12.3e\n", valor);
   printf ("%12.5e\n", valor);
 }
```

Quando você compilar e executar o programa *exp_fmt.c*, sua tela exibirá o seguinte:

```
    1.2e+00
   1.235e+00
 1.23456e+00
C:\>
```

70 JUSTIFICANDO À ESQUERDA A SAÍDA DE PRINTF

Por padrão quando você exibir o texto usando os caracteres de formatação, *printf* exibirá o texto justificado à direita. Dependendo do seu programa, algumas vezes você pode querer que *printf* justifique o texto à esquerda. Para justificar texto à esquerda, coloque um sinal de subtração (-) imediatamente após o % no especificador de formato. O programa a seguir, *esquerda.c*, ilustra o uso do sinal de subtração para justificar o texto à esquerda:

```
#include <stdio.h>

void main(void)
 {
   int int_valor = 5;
   float flt_valor = 3.33;

   printf("Justificado à direita %5d valor\n", int valor);
   printf("Justificado à esquerda %-5d valor\n", int_valor);
   printf("Justificado à direita %7.2f valor\n", flt_valor);
   printf("Justificado à esquerda %-7.2f valor\n", flt_valor);
 }
```

Quando você compilar e executar o programa *esquerda.c*, sua tela exibirá o seguinte:

```
Justificado à direita         5 valor
Justificado à esquerda 5       valor
Justificado à direita        3.33 valor
Justificado à esquerda  3.33    valor
C:\>
```

71 COMBINANDO OS ESPECIFICADORES DE FORMATO DE PRINTF

Várias dicas apresentadas nesta seção discutiram vários especificadores de formato de *printf*. À medida que você usar os especificadores de formato de *printf*, algumas vezes você poderá querer aproveitar dois ou mais especificadores de formato. Por exemplo, você pode querer exibir um valor hexadecimal justificado à esquerda, precedido pelos caracteres *0x*. Em tais casos, simplesmente coloque cada um dos especificadores após o %. O programa a seguir, *todo_fmt.c*, ilustra o uso de múltiplos especificadores de formato:

```
#include <stdio.h>

void main(void)
 {
   int int_valor = 5;
```

```
    printf("Justificado à esquerda com sinal %-+3d\n", int_valor);
}
```

Quando você compilar e executar o programa *todo_fmt.c*, sua tela exibirá o seguinte:

```
Justificado à esquerda com sinal  +5
C:\>
```

QUEBRANDO UMA STRING DE CARACTERES EM DUAS LINHAS 72

Quando seus programas usam *printf*, algumas vezes uma string de caracteres não cabe na linha corrente. Nesses casos, simplesmente coloque uma barra invertida (\) no final da linha, o que fará o texto continuar no início da próxima linha, como mostrado aqui:

```
printf("Esta linha é muito longa, e, portanto, não cabe na \
mesma linha.");
```

Nota: Se você quebrar o texto para a próxima linha, não inclua espaços no início da próxima linha de texto. Se existirem espaços, o compilador C os incluirá dentro da string

EXIBINDO STRINGS DO TIPO NEAR E FAR 73

A seção Memória, mais à frente discute os ponteiros *near* e *far* em detalhes. Resumidamente, os ponteiros *near* e *far* representam os endereços de variáveis dentro do espaço de memória do programa. Os programas que rodam dentro de sistemas operacionais mais antigos, tal como o MS-DOS, usam ponteiros *far* para aumentar o intervalo de endereços de memória que o programa pode usar para armazenar informações. Quando seus programas trabalham com ponteiros de string *far*, algumas vezes você quererá exibir o conteúdo da string usando *printf*. No entanto, como você aprendera posteriormente na seção Funções, o compilador gerará um erro se você passar um ponteiro *far* para uma função que espera um endereço *near*. Se você quiser exibir o conteúdo de uma string *far* (cujo início um ponteiro de string *far* indica) usando *printf*, você precisará dizer a *printf* que está usando um ponteiro *far*. Para fazer isso, coloque um *F* maiúsculo (de *far*) imediatamente após o % no especificador de formato, como mostrado aqui:

```
printf("%Fs\n, alguma_string_far);
```

Como *%Fs* diz a *printf* que você está usando um ponteiro *far*, a chamada da função está correta. De uma forma similar, você pode dizer a *printf* que está passando uma string *near* colocando um *N* maiúsculo no especificador de formato. No entanto, como *printf* espera strings *near* por padrão, os especificadores de formato *%Ns* e *%s* têm o mesmo resultado. O seguinte programa C, *near_far.c*, ilustra o uso de *%Fs* e *%Ns* dentro de *printf*:

```
#include <stdio.h>

void main(void)
 {
   char *titulo_near = "Bíblia do Programador C/C++, do Jamsa!";
   char far *titulo_far = "Bíblia do Programador C/C++, do Jamsa!";

   printf("Título do livro: %Ns\n", titulo_near);
   printf("Título do livro: %Fs\n", titulo_far);
 }
```

Nota: O Visual C++ não distingue entre os ponteiros near e far. Se você tentar compilar o programa near_far.c sob o Visual C++, o compilador retornará um erro. Para atualizar automaticamente seus programas para rodar sob o Visual C++, inclua o arquivo de cabeçalho windef.h dentro de seus programas

TRABALHANDO COM OS CARACTERES ESCAPE DE PRINTF 74

Quando você trabalha com strings de caracteres, algumas vezes quererá usar caracteres especiais, tais como tabulação, retorno do carro, ou alimentação de linha. C define vários *caracteres de escape* (isto é, caracteres que você

precede com o símbolo de escape de C, a barra invertida) para facilitar para você a inclusão de caracteres especiais dentro de uma string. Por exemplo, vários dos programas apresentados neste livro usam o caractere de nova linha (\n) para avançar a saída para o início da próxima linha, como mostrado aqui:

```
printf("Linha 1\nLinha2\nLinha 3\n");
```

A Tabela 74 lista os caracteres de escape que você pode usar dentro de suas strings de caractere (e, portanto, a saída de *printf*).

Tabela 74 Caracteres de escape definidos por C.

Caractere de Escape	Significado
\a	Caractere ASCII de aviso sonoro
\b	Caractere de retrocesso
\f	Caractere de avanço de formulário
\n	Caractere de nova linha
\r	Retorno do carro (sem alimentação de linha)
\t	Tabulação horizontal
\v	Tabulação vertical
**	Caractere de barra invertida
\'	Apóstrofe
\"	Aspas
\?	Interrogação
\nnn	Valor ASCII em octal
\xnnn	Valor ASCII em hexadecimal

75 DETERMINANDO O NÚMERO DE CARACTERES QUE PRINTF EXIBIU

Quando seus programas efetuam formatação sofisticada na tela, algumas vezes você quererá saber o número de caracteres que *printf* exibiu. Quando você usar o especificador de formato *%n*, *printf* atribuirá a uma variável (passada por ponteiro) um contador do número de caracteres que *printf* exibiu. O programa a seguir, *prt_cnt.c*, ilustra o uso do especificador de formato *%n*:

```
#include <stdio.h>

void main(void)
 {
   int primeiro_conta;
   int segundo_conta;

   printf("Bíblia%n do Programador C/C++, do Jamsa!%n\n", &primeiro_conta,
     &segundo_conta);
   printf("Primeiro conta %d Segundo conta %d\n", primeiro_conta,
     segundo_conta);
 }
```

Quando você compilar e executar o programa *prt_cnt.c*, sua tela exibirá o seguinte:

```
Bíblia do Programador C/C++, do Jamsa!
Primeiro conta 6 Segundo conta 38
C:\>
```

76 USANDO O VALOR DE RETORNO DE PRINTF

Na Dica 75 você aprendeu como usar o especificador de formato *%n* de *printf* para determinar o número de caracteres que *printf* escreveu. Usar o especificador de formato *%n* é um modo de garantir que *printf* teve sucesso

ao exibir sua saída. Além disso, quando *printf* termina, ele retorna o número total de caracteres que escreveu. Se *printf* encontrar um erro, retornará a constante EOF (que, como você aprenderá, indica o fim de um arquivo). O programa a seguir, *printfok.c*, usa o valor de retorno de *printf* para garantir que *printf* foi bem-sucedido:

```
#include <stdio.h>

void main(void)
 {
   int result;

   result = printf("Bíblia do Programador C/C++, do Jamsa!\n");
   if (result == EOF)
     fprintf(stderr, "Erro dentro de printf\n");
 }
```

Se o usuário tiver redirecionado a saída do programa para um arquivo ou um dispositivo (tal como uma impressora) e a E/S redirecionada experimentar um erro (tal como *dispositivo fora de linha* ou *disco cheio*), seus programas podem detectar o erro testando o valor de retorno de *printf*.

USANDO O CONTROLADOR DE DISPOSITIVO ANSI 77

Várias dicas apresentadas neste livro utilizam muito as capacidades de formatação de *printf*. Embora *printf* ofereça especificadores de formato que você usa para controlar o número de dígitos mostrados, para exibir saída em octal ou hexadecimal, ou para justificar texto à direita ou à esquerda, *printf* não oferece outros especificadores de formato. *printf* não fornece especificadores de formato que lhe permitem posicionar o cursor em uma linha e coluna específicas, limpar a tela ou exibir saída em cores. No entanto, dependendo do sistema operacional que estiver usando, você pode provavelmente efetuar essas operações usando o controlador de dispositivo ANSI. O controlador ANSI suporta diferentes sequências de escape que o instruem a usar cores específicas, a posicionar o cursor e até a limpar a tela. Os programadores chamam esses comandos de formatação de *sequências de escape*, pois elas iniciam com o caractere ASCII Escape (o valor 27). Se você estiver usando o DOS, instale o controlador de dispositivo colocando uma entrada tal como a seguinte dentro do seu arquivo *config.sys* (e depois reinicialize o computador):

```
DEVICE=C:\DOS\ANSI.SYS
```

Após você instalar o controlador ANSI, seus programas podem escrever seqüências de Escape usando *printf*.

Nota: Se você está rodando o Windows 95 na mesma máquina que compila programas, adicionar o controlador ANSI o seu arquivo **config.sys** *do sistema, não afetará as operações no Windows 95.*

USANDO O CONTROLADOR ANSI PARA LIMPAR SUA TELA 78

Uma das operações mais comuns que cada um dos seus programas executará quando iniciar sua execução é limpar a tela de vídeo. Infelizmente, a biblioteca de execução de C não fornece uma função que limpa a tela. Para limpar a tela, use o controlador ANSI descrito na Dica 77, e, depois, chame a seguinte sequência de escape para limpar sua tela de vídeo:

```
Esc[2J
```

Um modo fácil de chamar a sequência de escape é usar a representação octal do caractere escape (\033). Para imprimir o caractere escape, faça como mostrado aqui:

```
printf("\033[2J");
```

USANDO O CONTROLADOR ANSI PARA EXIBIR AS CORES DA TELA 79

Várias dicas apresentadas até aqui neste livro utilizam muito a função *printf* para exibir saída. Embora *printf* ofereça especificadores de formato poderosos, não oferece um modo para você exibir texto em cores. No entanto,

se estiver usando o controlador ANSI, como discutido na Dica 77, você pode usar as sequências de escape listadas na Tabela 79 para exibir a saída em cores.

Tabela 79 Seqüências ANSI que você pode usar para definir as cores da tela

Seqüência de Escape	Cor
Esc[30m	Cor do primeiro plano preta
Esc[31m	Cor do primeiro plano vermelha
Esc[32m	Cor do primeiro plano verde
Esc[33m	Cor do primeiro plano laranja
Esc[34m	Cor do primeiro plano azul
Esc[35m	Cor do primeiro plano magenta
Esc[36m	Cor do primeiro plano ciano
Esc[37m	Cor do primeiro plano branca
Esc[40m	Cor de fundo preta
Esc[41m	Cor de fundo vermelha
Esc[42m	Cor de fundo verde
Esc[43m	Cor de fundo laranja
Esc[44m	Cor de fundo azul
Esc[45m	Cor de fundo magenta
Esc[46m	Cor de fundo ciano
Esc[47m	Cor de fundo branca

O comando *printf* a seguir seleciona a cor de fundo azul:

```
printf("\033[44m");
```

Similarmente, o comando *printf* a seguir seleciona texto vermelho em um fundo branco:

```
printf("\033[47m\033[31m");
```

No exemplo anterior, *printf* escreve duas seqüências de escape. O controlador ANSI lhe permite especificar as cores da tela, separando-as com ponto-e-vírgula, como mostrado aqui:

```
printf("\033[47;31m");
```

80 USANDO O CONTROLADOR ANSI PARA POSICIONAR O CURSOR

Como você aprendeu, o controlador ANSI suporta as seqüências ANSI que, por sua vez, lhe permitem limpar a tela e exibir saída em cores. Além disso, o controlador ANSI oferece sequências de escape que lhe permitem posicionar o cursor para especificar as posições de linha e de coluna, que, por sua vez, lhe permitem exibir sua saída em posições específicas na tela. A Tabela 80 mostra as sequências de escape para o posicionamento do cursor.

Tabela 80 Seqüências de escape de posicionamento do cursor do controlador ansi que você pode usar em seus programas.

Seqüência de Escape	Função	Exemplo
Esc[x;yH	Posiciona o cursor na linha x, coluna y	*Esc[10;25H*
Esc[xA	Move o cursor *x* linhas para cima	*Esc[1a*
Esc[xB	Move o cursor *x* linhas para baixo	*Esc[2b*
Esc[yC	Move o cursor y colunas para a direita	*Esc[10c*
Esc[yD	Move o cursor *y* colunas para a esquerda	*Esc[10d*
Esc[S	Armazena a posição atual do cursor	*Esc[S*
Esc[U	Restaura a posição do cursor	Esc[U
Esc[2j	Limpa a tela, movendo o cursor para o canto superior esquerdo	*Esc[2j*
Esc[K	Limpa até o final da linha atual	*Esc[K*

EXECUTANDO OPERAÇÕES MATEMÁTICAS BÁSICAS EM C 81

Quase todos os seus programas, exceto os mais simples, efetuarão operações aritméticas, tais como adição, subtração, multiplicação ou divisão. Para efetuar essas operaçoes matemáticas básicas, use os operadores descritos na Tabela 81.

Tabela 81 Operadores aritméticos básicos em C.

Operador	Propósito
+	Adição
-	Subtração
*	Multiplicação
/	Divisão

O programa a seguir *matemat.c*, ilustra como usar os operadores aritméticos básicos de C:

```
#include <stdio.h>

void main(void)
 {
   int segundos_na_hora;
   float media;

   segundos_na_hora = 60 * 60;
   media = (5 + 10 + 15 + 20) / 4;
   printf("Número de segundos em uma hora %d\n", segundos_na_hora);
   printf("A média de 5, 10, 15, e 20 é %f\n", media);
   printf("O número de segundos em 48 minutos é %d\n",
        segundos_na_hora - 12 * 60);
 }
```

Quando você compilar e executar o programa *matemat.c*, sua tela exibirá o seguinte resultado:

```
Número de segundos em uma hora 3600
A média de 5, 10, 15, e 20 é 12.00000
O número de segundos em 48 minutos é 2880
C:\>
```

COMPREENDENDO A ARITMÉTICA DO MÓDULO 82

Na Dica 81 vimos que C usa o operador de barra para a frente (/) para a divisão. Dependendo do seu aplicativo, algumas vezes você precisará do resto de uma divisão inteira. Nesses casos, use o operador módulo (resto) de C. O programa a seguir, *modulo.c*, ilustra como usar o operador de módulo de C:

```
#include <stdio.h>

void main(void)
 {
   int resto;
   int result;

   result = 10 / 3;
   resto = 10 % 3;
   printf("10 dividido por 3 é %d Resto %d\n", result, resto);
 }
```

Quando você compilar e executar o programa *modulo.c*, sua tela exibirá o seguinte:

```
10 dividido por 3 é 3 Resto 1
```

83 COMPREENDENDO A PRECEDÊNCIA E A ASSOCIATIVIDADE DOS OPERADORES

Na Dica 81 você aprendeu que C usa os seguintes operadores, o sinal de adição (+) para adição; o hífen ( ) para subtração, o asterisco (*) para multiplicação; e a barra (/) para a divisão. Quando seus programas usam esses operadores dentro de expressões aritméticas, você deverá compreender a precedência dos operadores em C, que especifica a ordem em que C efetua as operações aritméticas. Por exemplo, considere a seguinte expressão:

```
result = 5 + 2 * 3;
```

Se você assumir que C efetua as operações da esquerda para a direita (a adição antes da multiplicação), o resultado da expressão será 21:

```
result = 5 + 2 * 3;
       = 7 * 3;
       = 21;
```

No entanto, se C efetuar as operações primeiro, o resultado será 11:

```
result = 5 + 2 * 3;
       = 5 + 6;
       = 11;
```

Para evitar o problema de resultados indeterminados, C define uma *precedência dos operadores*, que determina quais operações são executadas primeiro. A Tabela 83 ilustra a precedência dos operadores em C:

Tabela 83 Precedência dos operadores em C.

Precedência dos Operadores (do maior para o menor)										
()	[]	.	->							
+	–	+	-	*	&	!	~	(tipo)	sizeof	
*	/	%								
+	-									
>>	<<									
==	!=									
&										
^										
|										
&&										
||										
?:										
=	+=	-=	*=	/=	%=	&=	^=	|=	<<=	>>=
,										

Quando você criar uma expressão, C executará as operações com as precedências mais altas primeiro. Se dois operadores tiverem a mesma precedência, C efetuará as operações da esquerda para a direita.

84 FORÇANDO A ORDEM DE AVALIAÇÃO DOS OPERADORES

Como foi visto na Dica 83, C efetua as operações em uma expressão com base na precedência de cada operador dentro da expressão. Em muitos casos, a ordem que C usará para avaliar os operadores não é aquela que você quer. Por exemplo, considere a seguinte expressão - seu objetivo é calcular a média de três valores:

```
media = 5 + 10 + 15 / 3;
```

Matematicamente, a média dos três valores 5 10 e 15 é 10. No entanto, ao avaliar a expressão anterior, C atribuirá à variável *media* o valor 20, como mostrado aqui:

```
media = 5 + 10 + 15 / 3;
      = 5 + 10 + 5;
      = 15 + 5;
      = 20;
```

Se você examinar a tabela de precedência dos operadores (apresentada na Dica 83), verá que o operador de divisão (/) tem uma precedência mais alta que o operador de adição (+). Portanto, você precisa modificar a ordem em que C efetua as operações. Quando avalia uma expressão, C sempre efetua as operações que aparecem dentro de parênteses antes de efetuar as outras operações. Quando você agrupar os valores que quer somar dentro de parênteses, C calculará a média correta, como mostrado aqui:

```
media = (5 + 10 + 15) / 3;
      = (15 + 15) /3;c
      = (30) / 3;
      = 10;
```

C efetua as operações dentro dos parênteses com base em suas regras de precedência de operadores. Se uma expressão contiver múltiplas expressões dentro de múltiplos conjuntos de parênteses, C efetua as operações dentro dos parênteses mais internos primeiro, como mostrado aqui:

```
result = ((5 + 3) * 2) - 3;
       = ((8) * 2) - 3;
       = (16) - 3;
       = 13;
```

COMPREENDENDO O OPERADOR DE INCREMENTO DE C 85

Uma operação muito comum que os programas realizam é incrementar o valor atual de uma variável por 1. Por exemplo, o comando a seguir incrementa o valor da variável *conta* por 1:

```
conta = conta + 1;
```

Como as operações de incremento são tão comuns, C oferece uma notação resumida que você pode usar para incrementar as variáveis dentro de seus programas, chamada *operador de incremento*. O comando a seguir usa o operador de *incremento* para acrescentar 1 ao valor de *conta*:

```
conta++;
```

O programa a seguir, *0_a_100.c*, usa o operador de *incremento* para imprimir os valores de 0 a 100:

```
#include <stdio.h>

void main(void)
 {
   int valor = 0;

   while (valor <= 100)
     {
       printf("%d\n", valor);
       valor++;
     }
 }
```

C oferece um operador de *incremento* de *prefixo* e de *sufixo*. Os dois comandos a seguir incrementam a variável *conta* por 1:

```
conta++;
++conta;
```

O primeiro comando usa o operador de *incremento de sufixo* de C. O segundo comando usa o operador de *incremento de prefixo*. Você deve distinguir entre os dois operadores, pois C trata os operadores de *prefixo* e de *sufixo* de forma diferente. Quando você usar o operador de *incremento de sufixo*, C primeiro usará o valor da

variável, e, depois, efetuará a operação de *incremento*. Por outro lado, quando você usa o operador de *incremento de prefixo*, C primeiro incrementa o valor da variável e depois usa a variável. Para compreender melhor a diferença entre os operadores de *incremento de prefixo* e de *sufixo*, considere o seguinte programa, *presuf.c*, que usa ambos os operadores.

```
#include <stdio.h>

void main(void)
 {
   int valor = 1;

   printf("Usando sufixo %d\n", valor++);
   printf("Valor após o incremento %d\n", valor);
   valor = 1;
   printf("Usando prefixo %d\n", ++valor);
   printf("Valor após o incremento %d\n", valor);
 }
```

Quando você compilar e executar o programa *presuf.c*, sua tela exibirá o seguinte:

```
Usando sufixo 1
Valor após o incremento 2
Usando prefixo 2
Valor após o incremento 2
C:\>
```

Como você pode ver, ao usar o operador de *sufixo*, C primeiro usa o valor da variável (exibindo o valor 1), e, depois, incrementa a variável (resultando em 2). Quando você usa o operador de *prefixo*, C primeiro incrementa a variável (resultando em 2), e, depois, exibe o valor já incrementado.

86 COMPREENDENDO O OPERADOR DE DECREMENTO DE C

Da mesma forma como muitas vezes você quer incrementar o valor de uma variável, muitas vezes você quer decrementar o valor atual 1 de uma variável, como mostrado aqui:

```
conta = conta - 1;
```

Como as operações de decremento são tão comuns, C oferece uma notação abreviada que você pode usar para efetuar tais operações — o *operador de decremento* de C. O comando a seguir usa o operador de *decremento* para subtrair 1 do valor de *conta*:

```
conta--;
```

Como foi o caso com o operador de *incremento*, C fornece um operador de *decremento de prefixo* e um de *sufixo*. Os dois comandos a seguir decrementam a *variável conta* por 1.

```
conta--;
--conta;
```

O primeiro comando usa o operador de *decremento de sufixo* de C. O segundo comando usa o operador de *decremento de prefixo*. Você pode compreender a distinção entre os dois operadores, pois C trata os operadores de *prefixo* e de *sufixo* de forma diferente. Quando você usar o *operador de sufixo*, C primeiro usará o valor da variável, e, depois, efetuará a operação de decremento. Por outro lado, quando você usa o operador de *prefixo*, C primeiro incrementa o valor da variável e depois usa a variável. Para compreender melhor a diferença entre os operadores de *decremento de prefixo* e *de sufixo*, considere o seguinte programa, *sufpre.c*, que usa ambos os operadores.

```
#include <stdio.h>

void main(void)
 {
   int valor = 1;
```

```
    printf("Usando sufixo %d\n", valor--);
    printf("Valor após o decremento %d\n", valor);
    valor = 1;
    printf("Usando prefixo %d\n", --valor);
    printf("Valor após o decremento %d\n", valor);
  }
```

Quando você compilar e executar o programa *sufpre.c*, sua tela exibirá o seguinte:

```
Usando sufixo 1
Valor após o decremento 0
Usando prefixo 0
Valor após o decremento 0
C:\>
```

Como você vê, ao usar o operador de *decremento de sufixo*, C primeiro usa o valor da variável (exibindo o valor 1) e depois decrementa o valor da variável (produzindo o valor 1). Quando você usar o operador de *decremento de prefixo*, C primeiro decrementa a variável (resultando em 0) e depois exibe o valor já decrementado.

COMPREENDENDO UMA OPERAÇÃO OU BIT A BIT 87

À medida que a complexidade de seus programas for aumentando você verá que pode aumentar o desempenho do programa ou reduzir as necessidades de memória de um programa usando as *operações bit a bit*. As *operações bit a bit* manipulam os valores um ou mais bits ao mesmo tempo. Quando você precisar manipular um valor um único bit, pode utilizar o operador *OU bit a bit* da linguagem C (|). O operador *OU bit a bit* examina cada bit dentro de dois valores e gera um terceiro valor como resultado. Por exemplo, assuma que duas variáveis contenham os valores 3 e 4, cujos bits são respectivamente, 00000011 e 00000100. O operador *OU bit a bit* retorna o valor 7, como mostrado aqui:

```
3  00000011
4  00000100
   --------
7  00000111
```

No valor 3, os bits 0 e 1 têm um valor 1 e todos os outros bits têm um valor 0. No valor 4, o bit 2 tem um valor 1, e todos os outros bits têm o valor 0. O resultado de uma operação *OU* terá um valor 1 dentro de cada bit correspondente que tenha um valor 1 em um dos valores originais. Neste caso, o resultado tem um valor 1 nos bits 0, 1 e 2. O programa a seguir, *ou_bit.c*, ilustra como você usa o operador *OU bit a bit* da linguagem C:

```
#include <stdio.h>

void main(void)
 {
   printf("0 | 0 é %d\n", 0 | 0);
   printf("0 | 1 é %d\n", 0 | 1);
   printf("1 | 1 é %d\n", 1 | 1);
   printf("1 | 2 é %d\n", 1 | 2);
   printf("128 | 127 é %d\n", 128 | 127);
 }
```

Quando você compilar e executar o programa *ou_bit.c*, sua tela exibirá o seguinte:

```
0 | 0 é 0
0 | 1 é 1
1 | 1 é 1
1 | 2 é 3
128 | 127 é 255
C:\>
```

88 COMPREENDENDO UMA OPERAÇÃO E BIT A BIT

Como visto na Dica 87, você pode descobrir que é possível aumentar o desempenho do seu programa ou reduzir os requisitos de memória dele usando as operações bit a bit. As operações bit a bit manipulam valores um ou mais bits ao mesmo tempo. Quando precisar manipular dados um ou mais bits ao mesmo tempo, você pode utilizar o operador *E bit a bit* da linguagem C (&). O operador *E bit a bit* examina cada bit dentro de dois valores e gera um terceiro valor como resultado. Por exemplo, assuma que duas variáveis contenham os valores 5 e 7, cujos bits são, respectivamente, 00000101 e 00000111. O operador *E bit a bit* retorna o valor 5, como mostrado aqui:

```
5     00000101
7     00000111
      --------
5     00000101
```

Se um bit dentro de ambos os termos tiver um valor 1, o operador *E bit a bit* ligará o bit correspondente dentro do resultado. Se um bit dentro de um dos dois termos contém um valor 0, o operador *E bit a bit* deixará em 0 o bit correspondente dentro do resultado. Neste caso, os bits 0 e 2 contêm valores 1 em ambos os termos, de modo que o resultado tem valores 1 nos bits 0 e 2, e valores 0 nos bits restantes. O programa a seguir, *e_bit.c*, ilustra o uso do operador *E bit a bit* da linguagem C:

```
#include <stdio.h>

void main(void)
 {
   printf("0 & 0 é %d\n", 0 & 0);
   printf("0 & 1 é %d\n", 0 & 1);
   printf("1 & 1 é %d\n", 1 & 1);
   printf("1 & 2 é %d\n", 1 & 2);
   printf("15 & 127 é %d\n", 15 & 127);
 }
```

Quando você compilar e executar o programa *e_bit.c*, sua tela exibirá o seguinte:

```
0 & 0 é 0
0 & 1 é 0
1 & 1 é 1
1 & 2 é 0
15 & 127 é 15
C:\>
```

89 COMPREENDENDO UMA OPERAÇÃO OU EXCLUSIVO BIT A BIT

Como você já aprendeu, as operações bit a bit manipulam um ou mais bits ao mesmo tempo. Quando você estiver manipulando os bits de um valor, em algumas situações precisará utilizar o operador *OU exclusivo* (^), que examina os bits em dois valores e liga os bits do resultado com base na tabela verdade mostrada na Tabela 89.

Tabela 89 Resultados de uma operação **OU exclusivo bit a bit**

X	Y	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Suponha que duas variáveis contenham os valores 5 e 7, cujos bits são, respectivamente, 00000101 e 00000111. O operador *OU exclusivo bit a bit* retorna o valor 2, como mostrado aqui:

```
5     00000101
7     00000111
      --------
2     00000010
```

O programa a seguir, *ou_exclu.c*, ilustra o uso do operador *OU exclusivo bit a bit* da linguagem C:

```
#include <stdio.h>

void main(void)
 {
   printf("0 ^ 0 é %d\n", 0 ^ 0);
   printf("0 ^ 1 é %d\n", 0 ^ 1);
   printf("1 ^ 1 é %d\n", 1 ^ 1);
   printf("1 ^ 2 é %d\n", 1 ^ 2);
   printf("15 ^ 127 é %d\n", 15 ^ 127);
 }
```

Quando você compilar e executar o programa *ou_exclu.c*, sua tela exibirá o seguinte:

```
0 ^ 0 é 0
0 ^ 1 é 1
1 ^ 1 é 0
1 ^ 2 é 3
15 ^ 127 é 112
C:\>
```

COMPREENDENDO A OPERAÇÃO INVERSO BIT A BIT 90

Como já vimos, as operações bit a bit manipulam um ou mais bits ao mesmo tempo. Quando você precisar manipular bits, talvez seja necessário utilizar o operador *inverso bit a bit* da linguagem C (~). O operador *inverso bit a bit* examina cada bit dentro de um valor e produz um segundo valor como resultado. A operação *inverso bit a bit* torna cada bit que contém 1 no valor original em 0 no valor resultante, e torna cada bit que contém 0 no original em 1 no valor resultante. Como um exemplo, assuma que uma variável caractere sem sinal contenha o valor 15. Portanto, a operação *inverso bit a bit* retornaria 240, como mostrado aqui:

```
15   00001111
240  11110000
```

Como você pode ver, cada bit que estava ligado no valor original fica desligado no resultado, e cada bit que estava desligado no original é deixado ligado no resultado. O programa a seguir, *inv_bit.c*, ilustra o uso do operador *inverso bit a bit* da linguagem C:

```
#include <stdio.h>

void main(void)
 {
   int valor = 0xFF;

   printf("O inverso de %X é %X\n", valor, ~valor);
 }
```

Quando você compilar e executar o programa *inv_bit.c*, sua tela exibirá o seguinte:

```
O inverso de FF é FF00
C:\>
```

APLICANDO UMA OPERAÇÃO AO VALOR DE UMA VARIÁVEL 91

À medida que você efetuar operações aritméticas dentro de seus programas, verá que freqüentemente atribui a uma variável o resultado de uma expressão que inclui o valor atual da variável. Por exemplo, considere os seguintes comandos:

```
total = total + 100;
conta = conta - 5;
metade = metade / 2;
```

Para os casos em que um operador de atribuição atualiza uma variável com o resultado de uma operação com o valor atual da variável, C fornece uma técnica de abreviação para expressar a operação Em resumo, você coloca o operador antes do operador de atribuição. Quando você usa a técnica de atribuição abreviada, os comandos a seguir serão equivalentes aos três comandos que acabam de ser mostrados:

```
total += 100;
conta -= 5;
metade /= 2;
```

Quando você usar esta técnica abreviada, os comandos a seguir são equivalentes:

```
variável += 10;          variável = variável + 10;
variável <<= 2;          variável = variável << 2;
variável &= 0xFF;        variável = variável & 0xFF;
variável *= 1.05;        variável = variável * 1.05;
```

92 COMPREENDENDO O OPERADOR CONDICIONAL DE C

Como você aprenderá, o comando *if-else* de C examina uma condição e efetua um conjunto de operações se a condição for verdadeira e outro conjunto se a condição for falsa. De um modo similar, C fornece um operador condicional que examina uma condição, e, com base no resultado, verdadeiro ou falso, retorna um dentre dois valores. O formato do operador *condicional* é como segue:

```
(condição) ? resultadoverdadeiro: resultadofalso
```

Para compreender melhor o operador *condicional*, considere a seguinte condição, que testa se a nota em um teste é maior ou igual a 60. Se o valor for maior ou igual a 60, o comando atribuirá à variável um *resultado* um A, de "aprovado". Se o valor for menor que 60, o comando atribuirá à variável um *resultado* R, de "reprovado":

```
resultado = (nota >= 60) ? 'A': 'R';
```

O comando é similar ao seguinte comando *if-else:*

```
if (nota >= 60)
   resultado = 'A';
else
   resultado = 'R';
```

O seguinte comando *printf* exibe a string "*Aprovado*" ou "*Reprovado*" com base no teste da nota:

```
printf("Nota %d Resultado %s\n", nota, (nota >= 60) ? "Aprovado":
  "Reprovado");
```

Quando você usar o operador condicional de C para atribuir o resultado de uma condição a uma variável, poderá reduzir o número de comandos *if-else* que usa dentro de seus programas.

93 COMPREENDENDO O OPERADOR SIZEOF DE C

Quando seus programas declaram uma variável, o compilador C aloca memória para armazenar o valor da variável. Ao escrever programas que efetuem operações de *entrada/saída* ou aloquem memória para as listas dinâmicas, verá que é conveniente saber a quantidade de memória que seu programa alocou para uma variável específica. O operador *sizeof* de C retorna o número de bytes que uma variável ou tipo requer. O programa a seguir, *sizeof.c*, ilustra o uso do operador *sizeof.*

```
#include <stdio.h>

void main(void)
 {
  printf("As variáveis do tipo int usam %d bytes\n", sizeof(int));
  printf("As variáveis do tipo float usam %d bytes\n", sizeof(float));
  printf("As variáveis do tipo double usam %d bytes\n", sizeof(double));
  printf("As variáveis do tipo unsigned usam %d bytes\n", sizeof(unsigned));
  printf("As variáveis do tipo long usam %d bytes\n", sizeof(long));
 }
```

Dependendo do seu computador e do hardware do sistema, a saída produzida pelo programa *sizeof* poderá ser diferente Quando você usar o *Turbo C++ Lite*, o programa exibirá o seguinte:

```
As variáveis do tipo int usam 2 bytes
As variáveis do tipo float usam 4 bytes
As variáveis do tipo double usam 8 bytes
As variáveis do tipo unsigned usam 2 bytes
As variáveis do tipo long usam 4 bytes
C:\>
```

EFETUANDO UM DESLOCAMENTO BIT A BIT 94

Quando você trabalhar com valores no nível de bit, algumas operações comuns que você efetuará são *deslocamentos de bits*, ou para a direita (para longe do bit mais significativo) ou para a esquerda (em direção ao bit mais significativo). Para ajudar seus programas a efetuar os deslocamentos bit a bit, C oferece dois operadores *deslocamentos bit a bit:* um operador que desloca os bits para a direita (>>) e um que desloca os bits para a esquerda (<<). A expressão a seguir usa o operador de *deslocamento para a esquerda bit a bit* para deslocar os valores na variável *sinaliz* duas posições para a esquerda:

```
sinaliz = sinaliz << 2;
```

Suponha que a variável *sinaliz* contenha o valor 2, como mostrado aqui:

```
0000 0010
```

Quando você deslocar o valor duas casas para a esquerda, o resultado será 8, como mostrado aqui:

```
0000 1000
```

Quando você desloca valores para a esquerda, C preenche com zeros as posições dos bits menos significativos. No entanto, quando você desloca o valor para a direita, o valor que C coloca na posição do bit mais significativo depende do tipo da variável. Se a variável for sem sinal (isto é você a declarou no programa como um tipo *unsigned*), C preencherá com 0 o bit mais significativo durante a operação de *deslocamento para a direita*. No entanto, se a variável é de um tipo com sinal (em outras palavras, se você não declarou a variável como *unsigned*), C usará o valor 1 se o valor é atualmente negativo, ou 0 se o valor for positivo. O programa a seguir, *desloca.c*, ilustra o uso dos operadores *deslocamento para a direita* e *deslocamento para a esquerda bit a bit*.

```
#include <stdio.h>

void main(void)
 {
  unsigned u_val = 1;
  signed int valor = -1;

  printf ("%u (sem sinal) deslocado à esquerda 2 vezes é %u\n",
 u_val,
u_val << 2);
  printf ("%u (sem sinal) deslocado à direita 2 vezes é %u\n",
 u_val, u_val
>> 2);
  u_val = 65535;
  printf ("%u (sem sinal) deslocado à esquerda 2 vezes é %u\n",
 u_val,
u_val << 2);
  printf ("%u (sem sinal) deslocado à direita 2 vezes é %u\n",
 u_val, u_val
>> 2);
  printf ("%d (com sinal) deslocado à esquerda 2 vezes é %d\n",
 valor,
valor << 2);
  printf ("%d (com sinal) deslocado à direita 2 vezes é %d\n",
 valor, valor
>> 2);
 }
```

95 EFETUANDO UMA ROTAÇÃO BIT A BIT

Na Dica 94 você viu como usar os operadores de *deslocamento para a esquerda* e *para a direita* de C. Quando você realiza um operação de *deslocamento para a esquerda*, C preenche com zero o bit menos significativo. Por outro lado, quando você realiza uma operação de *deslocamento para a direita*, o valor que C coloca na posição do bit mais significativo depende do tipo do valor e do valor atual. À medida que você for trabalhando no nível de bit, algumas vezes poderá querer simplesmente rotacionar os bits, em vez de deslocá-los para a direita ou para a esquerda. Quando você rotaciona os bits para a esquerda, o bit mais significativo do valor torna-se o menos significativo, enquanto os outros bits movem-se uma posição para a direita. Quando você rotaciona os valores para a direita, o valor do bit menos significativo torna-se o mais significativo. Para ajudá-lo a rotacionar os bits, muitos compiladores C fornecem as funções *_rotl* e *_rotr*, que rotacionam os bits que compõem um valor sem sinal para a esquerda ou para a direita, como mostrado aqui:

```
#include <stdlib.h>

unsigned _rotl(unsigned valor, int conta);
unsigned _rotr(unsigned valor, int conta);
```

A variável *conta* especifica o número de vezes que você quer rotacionar o valor. O programa a seguir, *rotacao.c*, ilustra o uso das funções *_rotl* e *_rotr*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   unsigned valor = 1;

   printf("%u rotacionado à direita uma vez é %u\n", valor, _rotr(valor, 1));
   valor = 5;
   printf("%u rotacionado à direita duas vezes é %u\n", valor, _rotr(valor, 2));
   valor = 65534;
   printf("%u rotacionado à esquerda duas vezes é %u\n", valor, _rotl(valor, 2));
 }
```

Quando você compilar e executar o programa *rotacao.c*, sua tela exibirá o seguinte:

```
1 rotacionado à direita uma vez é 32768
5 rotacionado à direita duas vezes é 16385
65535 rotacionado à esquerda duas vezes é 65531
C:\>
```

Nota: Muitos compiladores C também oferecem as funções _lrotl e _lrotr, que rotacionam valores inteiros ***unsigned long*** *para a esquerda ou para a direita.*

96 COMPREENDENDO OS OPERADORES CONDICIONAIS

Todos os programas mostrados anteriormene neste livro iniciaram sua execução com a primeira instrução em *main*, e executaram cada instrução na ordem sequencial. À medida que seus programas se tornarem mais complexos, algumas vezes o programa precisará executar um conjunto de instruções se uma condição for verdadeira e, possivelmente, outras instruções se a condição for falsa. Por exemplo, seu programa poderia ter diferentes instruções para cada dia da semana. Quando um programa realiza (ou não realiza) instruções com base em uma condição específica, o programa está realizando *processamento condicional*. Para realizar processamento condicional, o programa avaliará uma condição que gera um resultado verdadeiro ou falso. Por exemplo, a condição *Hoje é Segunda-Feira* é verdadeira ou falsa. Para ajudar seus programas a realizar processamento condicional, C fornece os comandos *if*, *if-else* e *switch*. Várias dicas a seguir discutem esses comandos em detalhes.

COMPREENDENDO O PROCESSAMENTO ITERATIVO 97

Todos os programas apresentados aqui anteriormente executaram suas instruções somente uma vez. Em alguns casos, um programa pode ou não ter executado um conjunto de instruções com base no resultado de uma condição testada. À medida que seus programas se tornarem mais complexos, algumas vezes um programa precisa repetir o mesmo conjunto de instruções um número específico de vezes ou até que o programa atenda a uma condição específica. Por exemplo, se você estiver escrevendo um programa que calcula as classificações dos alunos, o programa precisa executar os mesmos passos para cada aluno na classe. Similarmente, se um programa exibir o conteúdo de um arquivo, ele lerá e exibirá cada linha do arquivo até encontrar o marcador de fim de arquivo. Quando os programas repetem um ou mais comandos até que uma condição seja encontrada, o programa está realizando *processamento iterativo*. Cada passagem que o programa faz pelos comandos que está repetindo é uma *iteração*. Para ajudar seus programas a realizar processamento iterativo, C providencia os comandos *for*, *while* e *do while*. Várias dicas apresentadas neste livro discutem os comandos *for*, *while* e *do while* em detalhes.

COMPREENDENDO COMO C REPRESENTA VERDADEIRO E FALSO 98

Várias dicas apresentadas nesta seção discutiram as construções *condicionais* e *iterativas* de C, que executam um conjunto de instruções se uma condição for verdadeira e, possivelmente, outro conjunto de instruções se a condição for falsa. À medida que você trabalhar com as construções condicionais e iterativas, é importante compreender como C representará um valor verdadeiro ou falso. C interpreta qualquer valor que não seja 0 como verdadeiro. Da mesma forma, o valor 0 representa falso. Portanto, a seguinte condição será avaliada como verdadeira:

```
if (1)
```

Muitos programadores inexperientes escrevem suas condições como mostrado aqui:

```
if (expressão != 0)    // Testa se uma expressão é verdadeira
```

Quando você quiser testar se uma condição é verdadeira, inclua a expressão como mostrado aqui.

```
if (expressão)
```

Quando a expressão é avaliada para um valor diferente de zero (verdadeira), C executa o comando que segue imediatamente a condição. Quando a expressão é avaliada para zero (falsa), C não executa o comando que segue imediatamente a condição. Os operadores que trabalham com verdadeiro e falso são operadores *Booleanos*. O resultado de uma expressão Booleana é sempre um valor verdadeiro ou falso.

TESTANDO UMA CONDIÇÃO COM IF 99

À medida que seus programas tornarem-se mais complexos, eles freqüentemente executarão um conjunto de comandos quando a condição for verdadeira, e outro conjunto de comandos quando a condição for falsa. Quando seu programa precisará executar um *processamento condicional*, você usará o comando *if* da linguagem C. O formato do comando *if* é como segue:

```
if (condição)
    comando;
```

A condição que o comando *if* avalia, precisa aparecer dentro de parênteses e ser verdadeira ou falsa. Quando a condição for verdadeira, C executará o comando que aparece imediatamente após a condição. Quando a condição for falsa, seu programa não executará o comando que segue a condição. Como um exemplo, o comando *if* a seguir testa se a variável *idade* é maior ou igual a 21. Se a condição for verdadeira, o programa não executará o comando *printf* e continuará sua execução no primeiro comando após *printf* (o comando de atribuição da altura):

```
if (idade >= 21)
    printf("A variável idade é igual ou maior a 21\n");
altura = 182;
```

100 COMPREENDENDO OS COMANDOS SIMPLES E COMPOSTOS

Quando seu programa executar o processamento condicional, algumas vezes ele executará um ou mais comandos quando uma condição for verdadeira, e, possivelmente, vários outros comandos se a condição for falsa. Da mesma forma, quando seu programa executar processamento iterativo, algumas vezes seu programa repetirá um comando; enquanto outras vezes, o programa poderá repetir vários comandos. Quando você executar processamento *condicional* e *iterativo*, C classificará os comandos como simples ou compostos. Um *comando simples* é um comando único, tal como a atribuição de uma variável ou uma chamada a *printf*. O comando *if* a seguir chama um comando simples (*printf*) quando a condiçao for verdadeira:

```
if (condição)
   printf("A condição é verdadeira\n");
```

Um *comando composto*, por outro lado, consiste de um ou mais comandos contidos dentro de abre e fecha chaves. O comando *if* a seguir ilustra um comando composto:

```
if (condição)
 {
    idade = 21;
    altura = 182;
    peso = 80;
 }
```

Quando seu programa precisar executar múltiplos comandos com base em uma condição, ou quando precisar repetir vários comandos, você usará um comando composto e colocará os comandos dentro de abre e fecha chaves.

101 TESTANDO A IGUALDADE

À medida que seus programas se tornarem mais complexos, eles compararão o valor de uma variável com condições conhecidas e determinarão quais comandos executar em seguida. Para tomar tais decisões, seus programas usarão os comandos *if* ou *switch*. Como você aprendeu na Dica 99, o formato do comando *if* é como segue:

```
if (condição)
    comando;
```

A maioria dos comandos *if* testará se o valor de uma variável é igual a um valor específico. Por exemplo, o comando a seguir testa se a variável *idade* contém o valor 21:

```
if (idade == 21)
   comando;
```

C usa o sinal de igual duplo (==) nos testes de igualdade. Quando você escrever testes de igualdade, use o sinal de igual duplo (==) e não o sinal de igual (=) que C usa para uma atribuição. Como veremos na Dica 112, se você usar o operador de atribuição (=) em vez do sinal de igual duplo, C considerará sua condição como uma sintaxe correta. Infelizmente, quando o comando for executado, C não testará se a variável é igual ao valor especificado. Em vez disso, C atribuirá o valor especificado à variável.

Nota: Dependendo do nível de advertência, seu compilador talvez exiba uma mensagem de aviso sobre a atribuição dentro da condição esperada.

Assim como algumas vezes seus programas precisam testar se um valor é igual a outro, C usa o símbolo != para testar a diferença. O comando a seguir testa se a variável *idade* não é igual a 21:

```
if (idade != 21)
    comando;
```

O programa a seguir, *ig_dif.c*, usa os testes de C para igualdade (==) e diferença (!=):

```
#include <stdio.h>

void main(void)
```

```
{
  int idade = 21;
  int altura = 182;
  if (idade == 21)
    printf("A idade do usuário é 21\n");
  if (idade != 21)
    printf("A idade do usuário não é 21\n");
  if (altura == 182)
    printf("A altura do usuário é 182\n");
  if (altura != 182)
    printf("A altura do usuário não é 182\n");
}
```

Quando você compilar e executar o programa *ig_dif.c*, sua tela exibirá o seguinte:

```
A idade do usuário é 21
A altura do usuário é 182
C:\>
```

Para compreender como usar os operadores de igualdade e de diferença, experimente o programa *ig_dif.c* alterando os valores das variáveis *idade* e *altura*.

EFETUANDO TESTES RELACIONAIS 102

À medida que seus programas ficarem mais complexos, algumas vezes você precisará testar se um valor é maior que outro, menor que outro, maior ou igual a outro ou menor ou igual a outro valor. Para ajudá-lo a realizar esses testes, C fornece um conjunto de *operadores relacionais*. A Tabela 102 lista os operadores relacionais de C.

Tabela 102 Operadores relacionais de C.

Operador	Função
>	Operador *maior que*
<	Operador *menor que*
>=	Operador *maior ou igual*
<=	Operador *menor ou igual*

O comando *if* a seguir usa o operador *maior ou igual* de C (>=) para testar se a variável inteira *idade* é superior a 20:

```
if (idade >= 21)
    printf ("A idade é superior a 20);
```

EXECUTANDO UMA OPERAÇÃO LÓGICA E PARA TESTAR DUAS CONDIÇÕES 103

Na Dica 99 você aprendeu como usar o comando *if* da linguagem C para testar condições dentro de seu programa. À medida que seus programas tornarem-se mais complexos, eles eventualmente testarão múltiplas condições. Por exemplo, você pode querer que um comando *if* teste se o usuário tem ou não um cachorro, e, em caso afirmativo, se esse cachorro é um dálmata. Nos casos em que você quer testar se duas condições são verdadeiras, use o operador *lógico E*. A linguagem C representa o operador logico E com dois sinais de ampersand (&&), como mostrado no seguinte comando *if*:

```
if ((usuario_tem_cachorro) && (cachorro == dalmata))
   {
      // Comandos
   }
```

Ao encontrar um comando *if* que usa o operador *lógico E* (&&), C avalia as condições da esquerda para a direita. Se você examinar os parênteses, verá que o comando *if* anterior tem a seguinte forma:

```
if (condição)
```

No exemplo a seguir, a condição é realmente duas condições conectadas pelo operador *lógico E:*

```
(usuario_tem_cachorro) && (cachorro == dalmata)
```

Para a condição resultante ser avaliada como verdadeira quando seus programas usarem o operador *lógico E*, ambas as condições precisam ser avaliadas como verdadeiras. Se uma das condições for falsa, a condição resultante será avaliada como falsa.

Muitas dicas apresentadas neste livro usarão o operador *lógico E*. Em cada caso, para garantir que cada expressão seja avaliada com a precedência de operador correta, os programas colocarão as condições dentro de parênteses.

Nota: Não confunda o operador lógico E (&&) com o operador bit a bit (&). O operador lógico E avalia duas expressões Booleanas (verdadeira ou falsa) e produz um resultado verdadeiro ou falso. O **operador** *E bit a bit, por outro lado, manipula bits (1s e 0s).*

104 EXECUTANDO UMA OPERAÇÃO LÓGICA OU PARA TESTAR DUAS CONDIÇÕES

Na Dica 99 você viu como usar o comando *if* para testar condições dentro de seus programas. À medida que seus programas se tornarem mais complexos, você precisará testar múltiplas condições. Por exemplo, você poderá querer que um comando *if* teste se um usuário tem um cachorro ou se ele tem um computador. Nos casos em que você quer testar se uma das condições é verdadeira (ou se ambas são verdadeiras), pode usarrá o operador *lógico OU*. C representa a *lógica OU* com duas barras verticais (|), como mostrado aqui:

```
if (usuario_tem_cachorro) || (usuario_tem_computador))
   {
      // Comandos
   }
```

Ao encontrar um comando *if* que usa o operador *lógico OU* (||), C avalia as condições da esquerda para a direita. Se você examinar os parênteses, verá que o comando *if* anterior está no seguinte formato:

```
if (condição)
```

Neste exemplo em particular, a condição é realmente duas condições conectadas pelo operador *lógico OU*, como mostrado aqui:

```
(usuario_tem_cachorro) || (usuario_tem_computador))
```

Para a condição resultante ser avaliada como verdadeira quando você usar o operador lógico OU, somente uma das duas condições precisará ser avaliada como verdadeira. Se uma das condições (ou ambas) forem verdadeiras, a condiçao resultante será avaliada como verdadeira. Se ambas as condições forem avaliadas como falsas, o resultado será falso.

Muitas dicas apresentadas neste livro usam o operador *lógico OU* (||). Em cada caso, para garantir que toda expressão será avaliada com a precedência correta de operador, os programas colocarão as condições dentro de parênteses.

Nota: Não confunda o operador lógico OU (||) da linguagem C com o operador OU bit a bit (|). O operador lógico OU avalia duas expressões Booleanas e produz um resultado verdadeiro ou falso. O operador OU bit a bit, por outro lado, manipula bits (1s e 0s).

105 EXECUTANDO UMA OPERAÇÃO LÓGICA NÃO (NOT)

Quando seus programas usam o comando *if* para realizar processamento condicional, o comando *if* avalia uma expressão que produz um resultado verdadeiro ou falso. Dependendo do processamento do seu programa, algumas vezes você somente irá querer que o programa execute um conjunto de comandos quando a condicional for avaliada como falsa. Por exemplo, assuma que você queira que um programa teste se o usuário tem um cachorro. Se o usuário não tem um cachorro, o programa deverá exibir uma mensagem dizendo ao usuário para

comprar um dálmata. Se o usuário tiver um cachorro, o programa não deverá fazer nada. Quando você quiser que seu programa execute um ou mais comandos quando uma condição for falsa, deverá usar o operador *lógico NÃO* da linguagem C, que é representado usando-se o sinal de exclamação (!). Considere o seguinte comando *if*:

```
if (! usuario_tem_cachorro)
    printf ("Compre um dálmata\n");
```

As condições que usam o operador *lógico NÃO* essencialmente dizem que, quando uma certa condição não for verdadeira (em outras palavras, quando a condição for avaliada como falsa), você deverá executar o comando *if* (ou comandos compostos). Várias dicas apresentadas neste livro usam o operador *lógico NÃO* dentro de condições.

ATRIBUINDO O RESULTADO DE UMA CONDIÇÃO 106

Várias dicas nesta seção apresentaram diferentes condições que são avaliadas como verdadeiras ou falsas dentro de um *if*, *while*, *for* ou outro comando. Além de permitir que você use condições dentro de estruturas de controle iterativo e condicional, C também lhe permite atribuir o resultado de uma condição a uma variável. Por exemplo, assuma que seu programa use o resultado da mesma condição mais de uma vez, como mostrado aqui:

```
if ((strlen(nome) < 100)  &&  (hoje == SEGUNDA))
  {
    // comandos
  }
else if (strlen(nome) < 100)  &&  (hoje == TERCA))
  {
    // comandos
  }
else if (strlen(nome) >= 100)
  {
    // comandos
  }
```

Como você pode ver, o programa usa a condição *(strlen(nome) < 100)* três vezes. Cada vez que a condição aparece, o programa chama a função *strlen*. Nos comandos anteriores, o programa poderia (dependendo do valor de *hoje*), chamar *strlen* três vezes. Os comandos a seguir atribuirão o resultado da condição (verdadeiro ou falso) à variável *nome_ok*, e, depois, repetidamente usarão a variável (e não a condição). Usar a variável em vez da condição, como mostrado aqui, melhora o desempenho do programa:

```
nome_ok = (strlen(nome) < 100);
if (nome_ok && (hoje == SEGUNDA))
  {
    // comandos
  }
else if (nome_ok && (hoje == TERCA))
  {
    // comandos
  }
else if (! nome_ok)
  {
    // comandos
  }
```

DECLARANDO VARIÁVEIS DENTRO DE COMANDOS COMPOSTOS 107

Na Dica 100 você aprendeu sobre a diferença entre comandos simples e compostos. Como visto, um comando composto é um ou mais comandos agrupados dentro de abre e fecha chaves. O laço *while* a seguir (que lê linhas de um arquivo e exibe as linhas em letras maiúsculas) ilustra um comando composto:

```
while (fgets(linha, sizeof(linha), pa))
  {
```

```
    strupr(linha);
    fputs(linha, stdout);
  }
```

À medida que seus programas vão ficando mais complexos, algumas vezes o processamento que eles executam dentro de um comando composto requererá o uso de uma ou mais variáveis cujos valores você usa somente dentro do laço (como pode ser o caso com as variáveis contadoras). Quando você usa variáveis contadoras, por exemplo, normalmente declara essas variáveis contadoras no início do seu programa, imediatamente após o comando *main*. No entanto, se você usar uma variável dentro de um comando composto, poderá declarar a variável no início do comando, como mostrado aqui:

```
if (condição)
  {
    int contador;
    float total;
    // Outros comandos
  }
```

Neste caso, o programa declara duas variáveis no início do comando composto. Dentro do comando composto, você pode usar essas duas variáveis exatamente como se as definisse no início do seu programa. No entanto, você não pode referenciar essas variáveis fora do abre e fecha chaves do comando composto. Uma vantagem de declarar variáveis dentro do comando composto é que outro programador quando for ler o código de seu programa compreenderá melhor como e quando usar uma variável. Várias dicas apresentadas posteriormente neste livro enfocam o *escopo* de uma variável, ou as localizações dentro do seu programa no qual o programa "conhece" uma variável. Como regra, você deve limitar o conhecimento de uma variável de um programa a somente aquelas localizações que usam a variável — em outras palavras, você deve limitar o escopo da variável. Declarar variáveis no início de um comando composto, como descrito nesta dica, limita o escopo da variável ao abre e fecha chaves do comando composto.

Nota: Se você declarar variáveis dentro de um comando composto que tem o mesmo nome que as variáveis que você definiu fora do comando, o compilador C usará as variáveis recém-declaradas dentro do comando composto, e as variáveis originais fora do comando.

108 USANDO RECUOS PARA MELHORAR A LEGIBILIDADE

À medida que você cria seus programas, um dos melhores modos de melhorar a legibilidade do programa é usar os recuos (ou endentação). Toda vez que seu programa usar uma chave (tal como no início de um comando composto), você deverá considerar a endentação do seu código em dois ou mais espaços. Por exemplo, considere o seguinte programa, *recuos.c*:

```
#include <stdio.h>

void main(void)
 {
   int idade = 10;
   int usuario_tem_cachorro = 0;   // 0 é falso

   if (idade == 10)
     {
       printf("Os cães são animais importantes\n");
       if (! usuario_tem_cachorro)
         printf("Compre um dálmata\n");
     }
   printf("Happy é um dálmata\n");
 }
```

Examinando a endentação, você pode rapidamente entender quais comandos estão relacionados (por exemplo, os comandos compostos). A endentação não é importante para o compilador. Para o compilador, o programa a seguir, *sem_rec.c*, é idêntico ao exemplo anterior:

```
#include <stdio.h>

void main(void)
 {
int idade = 10;
int usuario_tem_cachorro = 0;   // 0 é falso
if (idade == 10)
 {
printf("Os cães são animais importantes\n");
if (! usuario_tem_cachorro)
printf("Compre um dálmata\n");
 }
printf("Happy é um dálmata\n");
 }
```

Como você pode ver, a endentação torna o primeiro programa muito mais fácil de compreender, para você e para os outros programadores.

USANDO A VERIFICAÇÃO ESTENDIDA DE CTRL+BREAK 109

Quando você criar programas que usam o laço *for*, *while* e *do* para iteração e que rodam no ambiente DOS, algumas vezes é preciso usar a combinação de teclas Ctrl+Break para finalizar um programa que está preso em um laço infinito. Por padrão o DOS verifica um Ctrl+Break sempre que escreve na tela, no disco ou na impressora, ou ao ler um caractere do teclado. Se seu programa não efetua essas operações dentro do laço que você quer interromper, não é possível usar o comando Ctrl+Break para finalizar o processamento do programa. No entanto, quando usa o comando BREAK do DOS, você pode aumentar o número de operações que, ao ser completadas, instruem o DOS a verificar o pressionamento de Ctrl+Break. Os programadores chamam esse teste adicional de *verificação estendida de Ctrl+Break*. O comando BREAK a seguir habilita a verificação estendida de Ctrl+Break:

```
C:\>  BREAK ON   <Enter>
```

Se você quiser que o DOS habilite automaticamente a verificação estendida de Ctrl+Break assim que o sistema iniciar, coloque um comando BREAK=ON no seu arquivo *config.sys*. Como o DOS está efetuando uma verificação estendida de Ctrl+Break, o desempenho geral do seu sistema cairá ligeiramente. No entanto, como você está começando a programar, verá que sua possibilidade de encerrar um programa usando Ctrl+Break é mais importante que uma ligeira perda no desempenho.

TESTANDO VALORES DE PONTO FLUTUANTE 110

Várias dicas apresentadas nesta seção usam os comandos *if* e *while* para testar o valor de uma variável. Por exemplo, os comandos a seguir testam variáveis inteiras:

```
if (idade == 21)
    // comandos

if (altura > 182)
    // comandos
```

No entanto, ao trabalhar com valores em ponto flutuante, você precisa ter cuidado quando testar o valor de uma variável. Por exemplo, o comando a seguir testa uma variável em ponto flutuante chamada *imposto_vendas:*

```
if (imposto_vendas == 0.065)
   // comandos
```

Na Dica 51, você aprendeu sobre a precisão em ponto flutuante e sobre o fato de o computador precisar representar valores em ponto flutuante usando um número fixo de bits. É impossível para o computador representar todos os valores exatamente. No caso do comando *if* anterior, por exemplo, o computador pode representar o valor 0.065 como 0.0649999. Como resultado, o comando *if* nunca será avaliado como verdadeiro. Para evitar tais erros no seu programa, não teste valores em ponto flutuante exatos. Em vez disso, teste um intervalo aceitável de valores, como mostrado aqui:

```
if (fabs(imposto_vendas - 0.065) <= 0.0001)
   // comandos
```

No exemplo anterior, devido à diferença entre o valor na variável *imposto_vendas* e 0.065 ser menor ou igual a 0.0001, o programa considerará os valores iguais.

111 LAÇO DE REPETIÇÃO INFINITO

Já vimos que, os comandos *for*, *while* e *do while* lhe permitem repetir um ou mais comandos até que eles encontrem uma dada condição. Dependendo do seu programa, algumas vezes você quererá que o programa repita um laço infinitamente. Por exemplo um programa que detecta vazamento radioativo em um reator nuclear deve estar permanentemente em execução. Para manter seus programas em execução infinita, simplesmente coloque uma constante diferente de zero dentro do laço, como mostrado aqui:

```
while (1)
```

Como é possível usar um valor diferente de zero para forçar seus programas a ficar em execução perpetuamente, você pode querer definir constantes para melhorar a legibilidade do seu programa. Por exemplo, você poderia usar a constante *SEMPRE*, como mostrado aqui.

```
#define SEMPRE 1
while (SEMPRE)
```

Para criar um laço para o exemplo anterior da central nuclear, você poderia usar o seguinte:

```
#define DERRETENDO 0
while (1 DERRETENDO)
```

112 TESTANDO UMA ATRIBUIÇÃO

Como você aprendeu, C usa um sinal de igual como operador de atribuição e o sinal de igual dobrado para testar a igualdade, como mostrado aqui:

```
nota = 100;
if (nota == MAX)
  {
    // comandos
  }
```

No fragmento de código precedente, o primeiro comando atribui o valor 100 para a variável *nota*. Em seguida, o comando *if* testa o valor da variável. Para ajudá-lo a reduzir o número de comandos no seu programa, C lhe permite testar o resultado de uma atribuição. Por exemplo, o comando *if* a seguir combina a atribuição e a condição de teste anterior.

```
if ((nota = 100) == MAX)
  {
    // comandos
  }
```

Primeiro, C efetuará a expressão que os parênteses contêm, atribuindo o valor 100 à variável *nota*. Em seguida, C comparará o valor que você atribuiu à variável nota à constante *MAX*. Se você remover os parênteses, como mostrado aqui, C atribuirá um valor diferente e efetuará um teste diferente:

```
if (nota = 100 == MAX)
```

Sem os parênteses, C testará se o valor 100 é igual à constante *MAX*, e, em caso afirmativo, atribuirá o valor 1 (verdadeiro) à variável *nota*. Se o valor 100 não for igual a *MAX*, o comando atribuirá o valor 0 (falso) à variável *nota*.

Mais comumente você usará o teste de atribuição quando quiser testar o valor que uma função (tal como *fopen* ou *getchar*) retorna, como mostrado aqui:

```
if ((pa = fopen("CONFIG.SYS", "r")) == NULL)
  {
    // comandos
  }

if ((letra = getchar()) == 'A')
  {
    // comandos
  }
```

CUIDADO COM OS COMANDOS IF-IF-ELSE 113

Quando você usa comandos *if-else*, um erro de lógica pode causar problemas se você não controlar qual *else* corresponde a qual *if*. Por exemplo, considere o seguinte fragmento de código:

```
teste_nota = 100;
nivel_atual = 'B';
if (teste_nota >= 90)
   if (nivel_atual == 'A')
      printf("Outro A para um aluno A\n");
else
   printf("Você deveria ter se esforçado mais\n");
```

O primeiro comando *if* testa se a nota de um aluno no exame foi maior ou igual a 90. Em caso afirmativo, um segundo comando *if* testa se o aluno já tem um nível 'A', e, em caso afirmativo, imprime uma mensagem. Com base na endentação, você esperaria que o comando *else* exibisse sua mensagem se a nota de teste fosse menor que 90. Infelizmente, não é assim que o fragmento de código processa as condições. Quando você coloca um comando *else* dentro de seu programa, C associa o *else* com o primeiro comando *if* sem *else*. Embora a nota do aluno no exame seja 100, o fragmento de código anterior imprimirá a mensagem dizendo que o aluno deveria ter se esforçado mais. Em outras palavras, o fragmento executa os comandos mostrados aqui:

```
if (teste_nota >= 90)
   if (nivel_atual == 'A')
      printf("Outro A para um aluno A\n");
   else
      printf("Você deveria ter se esforçado mais\n");
```

Para evitar que C associe o comando *else* com o *if* errado, coloque o comando *if* dentro de chaves, formando um comando composto, como mostrado aqui:

```
if (teste_nota >= 90)
  {
    if (nivel_atual == 'A')
      printf("Outro A para um aluno A\n");
  }
  else
      printf("Você deveria ter se esforçado mais\n");
```

EXECUTANDO COMANDOS UM NÚMERO ESPECÍFICO DE VEZES 114

Uma operação que seus programas executarão comumente é repetir um conjunto de comandos um número específico de vezes. Por exemplo, você poderia querer calcular as notas de exame de 30 alunos, determinar as altas e baixas de 100 ações negociadas na Bolsa de Valores, ou soar três vezes o alto-falante interno do seu computador. Para ajudar seus programas a repetir um ou mais comandos um número específico de vezes, C fornece o comando *for*; basta implementar o comando *for* como mostrado aqui:

```
for (valor_inicial; condição_final; valor_incremento)
    comando;
```

Quando seu programa repetir comandos um número específico de vezes, você normalmente usará uma variável, chamada *variável de controle*, que contará o número de vezes que você executar os comandos. O comando *for* contém quatro seções. A seção *valor_inicial* atribui à variável de controle o valor inicial da variável, que é, na maioria das vezes, 0 ou 1. A seção *condição_final* normalmente testa o valor da variável de controle para determinar se o programa executou os comandos um número desejado de vezes. A seção *valor_incremento* normalmente adiciona o valor 1 para a variável de controle toda vez que os comandos são executados. Finalmente, a quarta seção do comando *for* é o comando ou comandos que você quer repetir. Como seu programa repetidamente executa o comando ou comandos que você especificou (volta para o início do comando), o comando *for* é normalmente chamado de *laço for*. Considere o seguinte comando *for*, que exibirá os números 1 a 10 na sua tela:

```
for (contador = 1; contador <= 10; contador++)
   printf("%d\n", contador);
```

No exemplo anterior, *contador* é a variável controladora do laço. Primeiro, o laço *for* atribui o valor 1 à variável. Segundo, o laço *for* testa imediatamente se o valor de *contador* é menor ou igual a 10 (a *condição final* do laço). Se *contador* for menor ou igual a 10, o laço *for* imediatamente executará o próximo comando, que, neste exemplo, é *printf*. Após o programa completar o comando *printf*, o laço *for* executa a expressão que você especificou na seção *valor_incremento* do laço. Neste caso, o laço *for* incrementa o valor de *contador* em 1. Em seguida, o laço *for* imediatamente executa o teste *valor_final*. Se o valor de *contador* for menor ou igual a 10, o laço continuará. Portanto, na primeira vez em que o laço é repetido, o comando *printf* exibirá o valor 1. Na segunda iteração, o valor de *contador* é 2, depois 3 e assim por diante. Após *printf* exibir o valor 10, a seção *valor_incremento* incrementará o valor de *contador*, tornando-o 11. Quando o laço *for* efetua o teste *valor_final*, você verá que o valor de *contador* não é mais menor ou igual a 10, de modo que o laço terminará e seu programa continuará o processamento imediatamente após o laço *for*.

Para compreender melhor o processamento do laço *for*, considere o seguinte programa, *testafor.c*:

```
#include <stdio.h>

void main(void)
 {
   int contador;

   for (contador = 1; contador <= 5; contador++)
    printf("%d ", contador);
   printf("\nIniciando o segundo laço\n");
   for (contador = 1; contador <= 10; contador++)
    printf("%d ", contador);
   printf("\nIniciando o terceiro laço\n");
   for (contador = 100; contador <= 5; contador++)
    printf("%d ", contador);
 }
```

Quando você compilar e executar o programa *testafor.c*, sua tela exibirá o seguinte:

```
1 2 3 4 5
Iniciando o segundo laço
1 2 3 4 5 6 7 8 9 10
Iniciando o terceiro laço
C:\>
```

Como você pode ver, o primeiro laço *for* exibe os números de 1 até 5. O segundo laço *for* exibe os valores de 1 até 10. O terceiro laço *for* não exibe nenhum valor. Se você examinar com atenção, verá que o programa inicialmente atribui à variável de controle do laço o valor 100. Quando o comando *for* testa o valor, o laço *for* atende imediatamente à condição final, de modo que o laço não é executado.

Todos os exemplos apresentados nesta dica usaram comandos simples no laço *for*. Se você precisar repetir mais de um comando, coloque os comandos dentro de abre e fecha chaves, formando um *comando composto*, como mostrado aqui:

```
for (i = 1; i <= 10; i++)
  {
    // comandos
  }
```

PARTES DO COMANDO FOR SÃO OPCIONAIS 115

Na Dica 114 você aprendeu que o comando *for* permite que seu programa repita um ou mais comandos um número específico de vezes. Como foi visto, o laço *for* usa três seções dentro do comando *for*: uma inicialização, um teste e um incremento (a quarta seção do laço contém os comandos que o laço *for* repete):

```
for (inicialização; teste; incremento)
```

Dependendo do seu programa, algumas vezes você pode não querer usar cada uma das seções do comando *for*. Por exemplo, se você já atribuiu o valor 0 à variável *conta*, pode pular a seção de inicialização do laço. Depois, para exibir os números de 0 até 999, seu laço conterá o seguinte:

```
for (; conta < 1000; conta++)
   printf(" %d", conta);
```

No entanto, se você omitir uma das seções do laço *for*, precisa incluir o ponto-e-vírgula correspondente. Por exemplo, o laço *for* a seguir omite as seções de inicialização e de incremento.

```
for (; conta < 1000; )
   printf(" %d", conta++);
```

Da mesma forma, o comando *for* a seguir ficará em execução perpetuamente:

```
for (;;)
    // comando
```

Embora o comando *for* forneça estas seções opcionais, seu programa pode tornar-se mais difícil de ler se você as omitir. Como regra, caso não precisar usar todas as três partes do comando *for*, você deverá usar uma construção de laço diferente, tal como o comando *while*.

DECREMENTANDO OS VALORES EM UM COMANDO FOR 116

Como você aprendeu, um comando *for* lhe permite repetir um ou mais comandos um número específico de vezes. As Dicas 114 e 115 apresentaram vários comandos *for*. Em cada caso, o laço *for* contou de 1 a 5, de 1 a 10, e assim por diante. O comando *for* também lhe permite decrementar a variável de controle. Por exemplo, o seguinte comando *for* conta regressivamente os números 10, 9, 8, e assim por diante, até 1:

```
for (contador = 10; contador >= 1; contador--)
    printf("%d ", contador");
```

Como você pode ver, o comando *for* precedente é aproximadamente o oposto dos comandos que você viu nas dicas anteriores. O laço inicializa a variável de controle *contador* com um valor alto, e, depois, decrementa *contador* por um cada vez que o laço é repetido.

O programa a seguir, *for_regr.c*, usa o comando *for* para contar regressivamente, primeiro de 5 a 1, e, depois, de 10 a 1.

```
#include <stdio.h>

void main(void)
 {
   int contador;
```

```
    for (contador = 5; contador >= 1; contador--)
     printf("%d ", contador);
     printf("\nIniciando o segundo laço\n");
     for (contador = 10; contador >= 1; contador--)
     printf("%d ", contador);
     printf("\nIniciando o terceiro laço\n");
   for (contador = 0; contador >= 1; contador--)
    printf("%d ", contador);
}
```

Quando você compilar e executar o programa *for_regr.c*, sua tela exibirá a seguinte saída

```
5 4 3 2 1
Iniciando o segundo laço
10 9 8 7 6 5 4 3 2 1
Iniciando o terceiro laço
C:\>
```

Como você pode ver, o terceiro laço não exibe nenhum valor. Neste exemplo, o comando *for* inicializa *contador* com um valor que é menor que o valor final de 1. Portanto, o laço termina imediatamente

117 CONTROLANDO O INCREMENTO DO LAÇO FOR

Como você aprendeu, o laço *for* permite que seus programas repitam um ou mais comandos um número específico de vezes. Nas dicas anteriores, cada laço *for* incrementou ou decrementou a variável de controle por 1. C, no entanto, isso lhe permite incrementar a variável por qualquer valor que você quiser. Por exemplo, o seguinte comando *for* incrementa a variável de controle *contador* por 10 a cada iteração do laço:

```
for (contador = 0; contador <= 100; contador += 10)
    printf ("%d\n", contador);
```

De um modo similar, os laços *for* anteriores inicializaram a variável de controle em 1 ou em 0. Assim como você pode definir a quantidade de incremento ou de decremento para qualquer tamanho que você queira, C, novamente, permite-lhe inicializar a variável para qualquer valor que você quiser. O programa a seguir, *for_dif.c*, usa diferentes valores de incremento e de decremento:

```
#include <stdio.h>

void main(void)
 {
   int contador;

   for (contador = -100; contador <= 100; contador += 5)
    printf("%d ", contador);
   printf("\nIniciando o segundo laço\n");
   for (contador = 100; contador >= -100; contador -= 25)
    printf("%d ", contador);
}
```

118 USANDO LAÇOS FOR COM VALORES CHAR E FLOAT

Como você aprendeu, o comando *for* permite que seus programas repitam um conjunto de comandos um número específico de vezes. Cada um dos comandos *for* apresentados nas dicas anteriores usaram somente valores do tipo *int*. No entanto, você pode usar caracteres e valores em ponto flutuante nos seus laços *for*. Por exemplo, o seguinte laço *for* exibe as letras do alfabeto:

```
for (letra = 'A'; letra <= 'Z'; letra++)
    printf("%c", letra);
```

Da mesma forma, o laço a seguir incrementa um valor de ponto flutuante em 0.5:

```
for (porcent = 0.0; porcent <= 100.0; porcent += 0.5);
    printf("%f\n", porcent);
```

O programa a seguir, *for_mais.c*, ilustra o uso de letras e valores em ponto flutuante em um laço *for*:

```
#include <stdio.h>

void main(void)
 {
   char letra;
   float porcent;
   for (letra = 'A'; letra <= 'Z'; letra++)
     putchar(letra);
   for (letra = 'z'; letra >= 'a'; letra--)
     putchar(letra);
   putchar('\n');
   for (porcent = 0.0; porcent < 1.0; porcent += 0.1)
     printf("%3.1f\n", porcent);
 }
```

COMPREENDENDO UM LAÇO NULL 119

Você já sabe que o laço *for* lhe permite repetir um ou mais comandos até que a variável de controle do laço atenda a uma certa condição. No passado, quando os programadores queriam que seus programas fizessem uma breve pausa, talvez para exibir alguma mensagem, eles colocavam um *laço NULO* ou "não faz nada" em seus programas. Por exemplo, o seguinte laço *for* não faz nada 100 vezes.

```
for (contador = 1; contador <= 100; contador++)
   ;   // Não faz nada
```

Quando você colocar um laço *NULO* em seus programas, C efetuará a inicialização do laço, e, depois, repetidamente, testará e incrementará a variável de controle até que a variável de controle atenda à condição final. O teste repetido do laço consome tempo do processador, o que faz o programa retardar. Se o programa precisar de um retardo maior, você poderá aumentar a condição final:

```
for (contador = 1; contador <= 10000; contador++)
    ;  // Não faz nada
```

Usar as técnicas de retardo, tais como o laço *NULO*, poderá causar problemas. Primeiro, se o programa estiver rodando em um computador 286, 386 ou 486, a duração do retardo diferirá simplesmente devido à diferença de velocidade entre os diferentes microprocessadores. Segundo, se o programa estiver rodando em um ambiente de multitarefa, tal como o Windows, OS/2, ou Unix, os laços "não fazem nada" consomem tempo que o processador poderia estar gastando fazendo trabalho importante em outro programa. Se seus programas usam esse retardo, veja as funções descritas na seção Data e Hora, mais à frente.

COMPREENDENDO O LAÇO INFINITO 120

Como você aprendeu, o laço *for* lhe permite repetir um ou mais comandos um número específico de vezes. Quando o laço *for* atender à condição final, seu programa continuará sua execução no comando que segue imediatamente. Quando você usa laços *for*, precisa garantir que o laço atenderá a sua condição final. Caso contrário, o laço continuará sua execução para sempre. Esses laços interminaveis são chamados *laços infinitos*. Na maioria dos casos, os laços *infinitos* ocorrem como resultado de erro na programação. Por exemplo, considere o seguinte laço:

```
for (i = 0; i < 100; i++)
 {
   printf("%d ", i);
   resultado = valor * --i;    // causa do erro
 }
```

Como você pode ver, o segundo comando do laço decrementa o valor da variável de controle *i*. Especificamente, o laço decrementa o valor para -1, e, depois incrementa o valor para 0. Como resultado, o valor nunca atinge 100, de modo que o laço não termina. Quando seu programa entra em um laço infinito, talvez você possa pressionar `Ctrl+C` para finalizar o programa. O programa a seguir, *infinito.c*, ilustra um laço infinito:

```
#include <stdio.h>

void main(void)
 {
   int i;
   int result = 0;
   int valor = 1;

   for (i = 0; i < 100; i++)
    {
      printf("%d ", i);
      result = valor * --i;
    }

   printf("Resultado %d\n", result);
 }
```

Quando você compilar e executar o programa *infinito.c*, ele repetidamente exibirá o valor 0. Para finalizar o programa, pressione `Ctrl+C`.

121 USANDO O OPERADOR VÍRGULA DE C DENTRO DE UM LAÇO FOR

Como você aprendeu, quando declara variáveis, C lhe permite declarar múltiplas variáveis do mesmo tipo separando os nomes das variáveis com vírgulas:

```
int idade, altura, peso;
```

Além disso, C lhe permite separar as inicializações de variáveis com vírgulas, como mostrado aqui:

```
int idade = 25, altura = 182, peso = 80;
```

De um modo similar, C lhe permite inicializar e incrementar múltiplas variáveis em um laço *for* separando as operações com vírgulas. Considere o laço a seguir, que trabalha com as variáveis *i* e *j*:

```
for (i = 0, j = 100; i <= 100; i++, j++)
   printf("i = %d j = %d\n", i, j);
```

Muito provavelmente você trabalhará com múltiplas variáveis em um laço *for* (também conhecido como *laços for embutidos*) em seus programas que trabalham com matrizes. Você aprenderá mais sobre as matrizes na seção Matrizes e Estruturas, posteriormente. O programa a seguir, *for_2var.C*, ilustra o uso do operador vírgula em um laço *for*:

```
#include <stdio.h>

void main(void)
 {
   int i, j;
```

```
  for (i = 0, j = 100; i <= 100; i++, j++)
    printf("i = %d j = %d\n", i, j);
}
```

EVITE MODIFICAR O VALOR DE UMA VARIÁVEL DE CONTROLE EM UM LAÇO FOR 122

Já vimos que o comando *for* lhe permite repetir um ou mais comandos um número específico de vezes. Para efetuar esse processamento, o laço *for* usa uma *variável de controle*, que funciona como um contador. Como regra, você não deve alterar o valor de uma variável de controle no comando do laço *for*. O único local em que o valor da variável de controle deve ser alterado é nas seções de inicialização e de incremento do laço *for*. Ao alterar o valor da variável de controle nos comandos do programa, você corre o grande risco de criar um laço infinito, tornando seus programas mais difíceis de ser compreendidos. No entanto, algumas vezes, você pode querer que o laço termine ou pule a iteração atual quando a variável de controle for igual a um certo valor. Em tais casos, use o comando *break* ou *continue*, discutidos em detalhes em outras dicas nesta seção.

REPETINDO UM OU MAIS COMANDOS USANDO UM LAÇO WHILE 123

Como você aprendeu, o comando *for* lhe permite repetir um ou mais comandos um número específico de vezes. No entanto, em muitos casos, seus programas precisam repetir um ou mais comandos até que o laço atenda a uma condição específica que não envolva necessariamente uma contagem. Por exemplo, se você escrever um programa que exiba o conteúdo de um arquivo na tela, irá querer que o programa exiba cada linha do arquivo. Na maioria dos casos, você não saberá de antemão quantas linhas o arquivo contém. Portanto, não pode usar um laço *for* para exibir, por exemplo, 100 linhas. O arquivo poderia conter mais ou menos linhas. Em vez disso, você quererá que o programa leia e exiba linhas até chegar ao final do arquivo. Para fazer isso, seus programas podem usar o laço *while*. Você formatará o laço *while* como segue:

```
while (condição)
    comando;
```

Ao encontrar um laço *while* no seu programa, C testa a condição especificada. Se a condição for verdadeira, C efetuará os comandos contidos no laço. Se o comando for falso, C continuará a execução do seu programa ao primeiro comando que segue. Um laço *while* pode repetir um único comando ou um comando composto delimitado por abre e fecha chaves, como mostrado aqui:

```
while (condição)
  {
    // Comandos
  }
```

O programa a seguir, *espera.c*, usa o laço *while* para repetir um laço até que você pressione a tecla S ou N em resposta a uma pergunta:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main(void)
 {
   char letra;   // Letra digitada pelo usuário

   printf("Quer continuar? (S/N): ");

   letra = getch();         // Lê a letra
   letra = toupper(letra); // Converte a letra para maiúscula

   while ((letra != 'S') && (letra != 'N'))
     {
```

```
        putch(7);              // Soa o alto-falante
        letra = getch();       // Lê a letra
        letra = toupper(letra); // Converte a letra para maiúscula
      }

   printf("\nSua resposta foi %c\n", letra);
}
```

Primeiro, o programa exibirá a mensagem que o primeiro comando *printf* contém. Segundo, o programa usa *getch* para ler a tecla pressionada. Para simplificar o teste do laço, o programa converte a letra para maiúscula, de modo que o laço somente precisa testar as letras S ou N. Terceiro, o laço *while* testará a letra que o usuário digitou. Se a letra for um S ou N, a condição falhará, e os comandos do laço não serão executados. Se a letra pressionada não for S ou N, a condição do laço será verdadeira e seus comandos serão executados. No laço, o comando soará o alto-falante interno do computador para indicar um caractere inválido. Em seguida, o programa lerá a nova tecla e converterá a letra para maiúscula. O laço depois repetirá seu teste para determinar se o usuário digitou um S ou um N. Se não, os comandos do laço serão repetidos. Caso contrário, a execução do programa continuará no primeiro comando que segue o laço.

124 COMPREENDENDO AS PARTES DE UM LAÇO WHILE

Um laço *while* lhe permite executar um ou mais comandos até que o programa atenda à condição do laço. Na Dica 114, você aprendeu que um laço *for* realmente contém quatro seções: uma inicialização, um teste, um comando de execução e um incremento. Por outro lado, um laço *while* contém somente um teste e os comandos que você quer repetir, como mostrado aqui:

```
while (condição)
   comando;
```

Você viu na Dica 120 que um laço infinito é um laço cuja condição final nunca é atendida, e, portanto o laço continua a execução para sempre. Ao escrever programas que usam laços *while*, você pode reduzir a possibilidade de um laço infinito garantindo que seus laços *while* efetuem os mesmos passos executados por um laço *for*. Para ajudá-lo a lembrar os quatro passos, memorize a sigla *ITEM*, como ilustrado na Tabela 124:

Tabela 124 Os Componentes da Sigla ITEM.

Ação	Descrição
Inicializa	Inicializa a variável de controle do laço
Testa	Testa a variável de controle ou a condição do laço
Executa	Executa os comandos desejados no laço
Modifica	Modifica o valor da variável de controle ou efetua um operação que afetará a condição que você está testando

Ao contrário do laço *for*, que lhe permite explicitamente inicializar e incrementar uma variável de controle, um laço *while* requer que você inclua comandos no programa que efetuem esses passos para você. O programa a seguir, *item.c*, ilustra como seu programa efetua esses quatros passos. Ao contrário dos programas anteriores que você escreveu, *item.c*, usa um laço *while* para exibir os números de 1 a 100:

```
#include <stdio.h>

void main(void)
 {
   int contador = 1;  // Inicializa a variável de controle

   while (contador <= 100)  // Testa a variável de controle
    {
      printf("%d ", contador);  // Executa os comandos
      contador++;              // Modifica a variável de controle
    }
 }
```

Se você escrever um programa que usa o laço *while* e o programa ficar preso em um laço infinito, uma das operações ITEM no seu programa não está correta.

REPETINDO UM OU MAIS COMANDOS USANDO DO 125

Você já sabe que o comando *while* lhe permite repetir um ou mais comandos enquanto uma condição específica é atendida. Da mesma forma, o comando *for* lhe permite repetir um ou mais comandos um número específico de vezes. Além disso, C fornece o comando *do*, que lhe permite executar um ou mais comandos pelo menos uma vez, e, depois, se necessário, repetir comandos. O formato do comando *do* é como segue:

```
do
   comandos;
while (condição);
```

O comando *do* é ideal para situações que requerem que você execute um ou mais comandos pelo menos uma vez. Por exemplo, considere o seguinte fragmento de código:

```
printf("Quer continuar? (S/N): ");

letra = getch();           // Recebe o caractere
letra = toupper(letra);    // Converte a letra para maiúscula

while ((letra != 'S') && (letra != 'N'))
 {
   putch(7);                // Emite um aviso sonoro
   letra = getch();         // Recebe o caractere
   letra = toupper(letra); // Converte a letra para maiúscula
 }
```

Como você pode ver, o código pede que o usuário digite uma tecla, lê a tecla e converte-a para maiúsculas. Dependendo da tecla que o usuário pressionou, o fragmento iniciará um laço *while*, que efetua os mesmos comandos. Observe que você pode simplificar os comandos usando o comando *do*, como mostrado no seguinte fragmento de código:

```
printf("Quer continuar? (S/N): ");
do
  {
    letra = getch();           // Lê a letra
    letra = toupper(letra);    // Converte a letra para maiúscula
    if ((letra != 'S') && (letra != 'N'))
       putch(7);    // Soa o alarme - caractere inválido
  }
while ((letra != 'S') && (letra != 'N'));
```

Ao encontrar um comando *do* no seu programa, C executa os comandos entre as palavras *do* e *while*. C então testa a condição que *while* especifica para determinar se os comandos devem ou não ser repetidos. Portanto, os comandos que um laço *do* especifica sempre são executados pelo menos uma vez. Os programas normalmente usam o laço *do* para exibir e processar opções do menu. O programa a seguir, *do_menu.c*, usa o comando para exibir e processar opções de menu até que o usuário selecione a opção Sair:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>

void main(void)
 {
   char letra;

   do
     {
```

```
    printf("A Exibir a listagem do diretório\n");
    printf("B Modificar a hora do sistema\n");
    printf("C Modificar a data do sistema\n");
    printf("S Sair\n");
    printf("Escolha: ");

    letra = getch();
    letra = toupper(letra);

    if (letra == 'A')
      system("DIR");
    else if (letra == 'B')
      system("TIME");
    else if (letra == 'C')
      system("DATE");
   }
  while (letra != 'S');
}
```

126 COMPREENDENDO O COMANDO CONTINUE DE C

Como você aprendeu, os comandos *for*, *while* e *do* permitem que seus programas repitam um ou mais comandos até que uma condição específica seja avaliada como verdadeira ou falsa. Dependendo do propósito do seu programa, algumas vezes, com base em uma segunda condição específica, você quererá que seu programa pule a iteração atual. O comando *continue* de C lhe permite fazer exatamente isso. Se C encontrar um comando *continue* em um laço *for*, C automaticamente executará a porção de incremento do laço, e, depois, efetuará o teste da condição final. Se C encontrar um comando *continue* em um laço *while* ou *do*, então C imediatamente efetuará o teste da condição final. Para compreender melhor o comando *continue*, considere o seguinte programa, *par_impar.c*, que usa *continue* em um laço *for* e um laço *while* para exibir os números pares e ímpares entre 1 e 100:

```
#include <stdio.h>

void main(void)
 {
   int contador;

   printf("\nValores pares\n");
   for (contador = 1; contador <= 100; contador++)
     {
       if (contador % 2)  // Ímpar
         continue;
       printf("%d ", contador);
     }
   printf("\nValores ímpares\n");
   contador = 0;
   while (contador <= 100)
    {
       contador++;
       if (! (contador % 2)) // Par
         continue;
       printf("%d ", contador);
     }
}
```

O programa usa o operador *módulo* (resto) para determinar se um valor é par ou ímpar. Se você dividir um valor por 2 e obtiver um resto de 1, o valor é ímpar. Da mesma forma, se obtiver um resto de 0, o valor é par.

É importante observar que você normalmente pode eliminar a necessidade de usar um comando *continue* reprojetando o uso dos comandos *if* e *else* em seu programa. Por exemplo, o programa a seguir *semcont.c*, também exibe valores pares e ímpares sem usar *continue:*

```
#include <stdio.h>

void main(void)
 {
   int contador;

   printf("\nValores pares\n");
   for (contador = 1; contador <= 100; contador++)
     {
       if (!(contador % 2))  // Par
         printf("%d ", contador);
     }
   printf("\nValores ímpares\n");
   contador = 0;
   while (contador <= 100)
     {
       contador++;
       if (contador % 2) // Ímpar
         printf("%d ", contador);
     }
 }
```

Antes de usar um comando *continue* no seu programa, examine seu código atentamente para determinar se você pode escrever os mesmos comandos sem usar *continue*. Na maioria dos casos, você verá que o código sem *continue* é mais fácil de compreender.

FINALIZANDO UM LAÇO USANDO O COMANDO BREAK DE C 127

Já vimos que os comandos *for*, *while* e *do* permitem que seus programas repitam um ou mais comandos até que uma condição específica seja avaliada como verdadeira ou como falsa. Dependendo do propósito do seu programa, algumas vezes, com base em uma segunda condição específica, você quererá que o laço termine automaticamente, com seu programa continuando seu processamento no comando que segue o laço. O comando *break* lhe permite fazer exatamente isso. Quando C encontra um break em um laço, a execução do laço terminará de imediato. O próximo comando que o programa executa é o comando que segue imediatamente o laço. No caso de um comando *for*, C não efetuará a seção de incremento do laço — em vez disso, o laço termina imediatamente. O programa a seguir, *usabreak.c*, ilustra o uso do comando *break*. O programa percorre os números de 1 a 100, e, depois, de 100 a 1. Toda vez que o laço chega ao valor 50, o comando *break* termina imediatamente o laço:

```
#include <stdio.h>

void main(void)
 {
   int contador;

   for (contador = 1; contador <= 100; contador++)
    {
      if (contador == 50)
        break;
      printf("%d ", contador);
    }
   printf("\nPróximo laço\n");
   for (contador = 100; contador >= 1; contador--)
    {
```

```
      if (contador == 50)
        break;
      printf("%d ", contador);
    }
}
```

Como foi o caso com o comando *continue*, você normalmente reescreverá as condições do laço e de *if-else* para eliminar a necessidade do comando *break* em laços. Na maioria dos casos, quando você reescrever os comandos do seu programa para eliminar *break*, seu programa ficará muito mais fácil para o leitor compreender. Como regra, limite o uso de *break* dentro do comando *switch*.

128 DESVIOS COM O COMANDO GOTO

Se você já programou em BASIC, FORTRAN ou em linguagem Assembly, pode estar habituado a implementar as operações *if-else* e os laços usando o comando *GOTO* Como a maioria das linguagens de programação, C fornece um comando *goto*, que permite que a execução do seu programa desvie-se para uma localização específica, chamada *rótulo*. O formato do comando *goto* é como segue:

```
goto rotulo;
rotulo:
```

O programa C a seguir, *goto_100.c*, usa o comando *goto* para exibir os números de 1 a 100:

```
#include <stdio.h>

void main(void)
 {
   int conta = 1;

   rotulo:
     printf("%d ", conta++);

     if (conta <= 100)
       goto rotulo;
 }
```

Quando você usa o comando *goto*, o rótulo precisa residir na função atual. Em outras palavras, você não pode usar *goto* para desviar-se de *main* para um rótulo que aparece em outra função, ou vice-versa.

Como no passado os programadores abusaram do comando *goto*, você deverá restringir o uso de *goto* sempre que for possível, e, em lugar dele, usar construções tais como *if*, *if-else* e *while* Na maioria dos casos, você poderá usar essas três construções para reescrever um fragmento de código que usa *goto*, e, desse modo, produzir um código mais legível.

129 TESTANDO MÚLTIPLAS CONDIÇÕES

Como foi visto, os comandos *if-else* de C lhe permitem testar múltiplas condições. Por exemplo, considere o seguinte teste da variável *letra*:

```
letra = getch();
letra = toupper(letra);

if (letra == 'A')
    system("DIR");
else if (letra == 'B')
    system("TIME");
else if (letra == 'C')
    system("DATE");
```

Nos casos em que você está comparando a mesma variável com múltiplos valores, C fornece um comando *switch*, com o seguinte formato:

```
switch (expressão)  {
    case Constante_1:  comando;
    case Constante_2:  comando;
    case Constante_3:  comando;
      :     :     :
};
```

Em vez de usar os comandos *if else* anterior, você poderia usar *switch* como segue:

```
switch (letra)  {
  case 'A':  system("DIR");
             break;
  case 'B':  system("TIME");
             break;
  case 'C':  system("DATE");
             break;
};
```

Ao encontrar um comando *switch* no seu programa, C avalia a expressão que segue para produzir um resultado. C então compara o resultado com cada um dos valores constantes, especificados por você, que seguem a palavra-chave *case*. Se C encontrar uma coincidência, ele executará os comandos correspondentes. O comando *break* separará comandos correspondentes de um *case* para o outro. Você normalmente colocará um comando *break* após o último comando que corresponde a uma opção. Na Dica 130, você aprenderá os detalhes que governam o uso do comando *break* em *switch*. O programa a seguir, *swt_menu.c*, usa o comando *switch* para processar a seleção de menu de um usuário:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <stdlib.h>

void main(void)
 {
   char letra;

   do {
     printf("A Exibir listagem do diretório\n");
     printf("B Alterar a hora do sistema\n");
     printf("C Alterar a data do sistema\n");
     printf("S Sair\n");
     printf("Escolha: ");
     letra = getch();
     letra = toupper(letra);
     switch (letra) {
       case 'A': system("DIR");
                 break;
       case 'B': system("TIME");
                 break;
       case 'C': system("DATE");
                 break;
     };
    }
   while (letra != 'S');
 }
```

130 COMPREENDENDO BREAK DENTRO DE SWITCH

Na Dica 129 você aprendeu que o comando *switch* de C lhe permite a execução de processamento condicional. Assim, você especifica um ou mais *cases* correspondentes possíveis usando o comando *switch*. Para cada *case*, você especifica os comandos correspondentes. Ao final dos comandos, você normalmente coloca um comando *break* para separar um comando *case* de outro. Se você omitir o comando *break*, C continuará a executar todos os comandos seguintes, independente do *case* ao qual o comando pertence. Por exemplo, considere o seguinte comando *switch*:

```
switch (letra)   {
    case 'A':  system("DIR");
    case 'B':  system("TIME");
    case 'C':  system("DATE");
};
```

Se a variável *letra* contiver a letra A, C executará o comando DIR. No entanto, como não existe um comando *break* em seguida, o programa também executará os comandos TIME e DATE. Se a variável *letra* contivesse a letra B, o programa executaria os comandos TIME e DATE. Para evitar as execução dos comandos dos outros *cases*, use o comando *break*, como mostrado aqui:

```
switch (letra)  {
    case 'A':  system("DIR");
               break;
    case 'B':  system("TIME");
               break;
    case 'C':  system("DATE");
               break;
};
```

Algumas vezes você poderá querer que seus programas testem em seqüência várias opções. Por exemplo, o programa a seguir, *vogais.c*, usa um comando *switch* para contar o número de vogais no alfabeto:

```
#include <stdio.h>

void main(void)
{
  char letra;
  int conta_vogal = 0;

  for (letra = 'A'; letra <= 'Z'; letra++)
    switch (letra) {
      case 'A':
      case 'E':
      case 'I':
      case 'O':
      case 'U': conta_vogal++;
    };
  printf("O número de vogais é %d\n", conta_vogal);
}
```

Neste caso, se a variável letra contém o A, E, I ou O, a ocorrência que executa o comando corresponde à letra U, que incrementa a variável *conta_vogal*. Como o comando *switch* não contém nenhum outro *case* após a letra U, o programa não inclui o comando *break*.

131 USANDO O CASE DEFAULT DO COMANDO SWITCH

Vimos que o comando *switch* de C lhe permite efetuar um processamento condicional. Ao usar o comando *switch*, você especifica um ou mais *cases* que deseja que C confira, como mostrado aqui:

```
switch (letra)  {
  case 'A': system("DIR");
            break;
  case 'B': system("TIME");
            break;
  case 'C': system("DATE");
            break;
};
```

À medida que você usar o comando *switch*, é possível ver que algumas vezes você desejará que C efetue comandos específicos quando os outros *cases* não conferirem. Para fazer isso, inclua um *case default* com o comando *switch*, como mostrado aqui:

```
switch (expressão)
   case constante_1: comando;
   case constante_2: comando;
   case constante_3: comando;
    :     :           :
   default:   comando;
};
```

Se C não executar nenhumas das opções *case* que precedem o *default*, C executará os comandos de *default*. O programa a seguir, *cons_vog.c*, usa o *case default* para controlar o numero de letras consoantes no alfabeto:

```
#include <stdio.h>

void main(void)
{
   char letra;
   int conta_vogal = 0;
   int conta_consoante = 0;

   for (letra = 'A'; letra <= 'Z'; letra++)
     switch (letra)  {
       case 'A':
       case 'E':
       case 'I':
       case 'O':
       case 'U': conta_vogal++;
                 break;
       default: conta_consoante++;
     };
   printf("O número de vogais é %d\n", conta_vogal);
   printf("O número de vogais é %d\n", conta_consoante);
}
```

DEFININDO CONSTANTES NO SEU PROGRAMA 132

Como regra, você pode aumentar a legibilidade e portabilidade do seu programa substituindo as referências por números, tal como 512, com um nome de constante mais representativo. Uma *constante* é um nome que o compilador C associa com um valor que não muda. Para criar uma constante, você usa a diretiva *#define*. Por exemplo, a diretiva a seguir cria uma constante chamada *TAM_LINHA*, e atribui à constante o valor 128:

```
#define TAM_LINHA 128
```

Quando o pré-processador C encontra o nome de constante *TAM_LIN* no seu programa, ele substitui o nome da constante pelo valor da constante. Por exemplo, considere a seguinte declaração de string de caracteres:

```
char linha[128];
```

```
char texto[128];

char linha_atual[TAM_LINHA];
char entrada_usuario[TAM_LINHA];
```

As duas primeiras declarações criam strings de caracteres que contêm strings de 128 caracteres. As duas declarações a seguir criam strings de caracteres que estão baseadas em uma constante chamada TAM_LINHA. Quando outros programadores lerem o código do seu programa, uma das primeiras perguntas que eles poderão fazer é por que você usou 128 na declaração da sua string. No entanto, no caso da segunda declaração, o programador sabe que você declarou todas as suas strings em termos de um *TAM_LINHA* predefinido. Dentro do seu programa, você poderia incluir laços similares ao seguinte:

```
for (i=0; i < 128; i++)
   // comandos

for (i=0; i < TAM_LINHA; i++)
   // comandos
```

O segundo laço *for* torna seu programa mais legível e fácil de alterar. Por exemplo, assuma que seu programa use o valor 128 para referenciar o tamanho da string. Se, mais tarde, você quiser alterar o tamanho para 256 caracteres, precisará alterar cada ocorrência do valor 128 no seu programa — um processo demorado. Por outro lado, se você estiver usando uma constante tal como *TAM_LINHA*, somente precisará alterar a diretiva *#define* — um processo de uma única etapa —, como mostrado aqui:

```
#define TAM_LINHA 256
```

133 COMPREENDENDO A EXPANSÃO DE MACROS E DE CONSTANTES

Na Dica 132, você aprendeu que seus programas podem usar a diretiva *#define* para definir uma constante dentro de seu programa. O programa a seguir, *ex_macro.c*, por exemplo, usa três constantes:

```
#define LINHA 128
#define TITULO "Bíblia do Programador C/C++, do Jamsa!"
#define SECAO "Macros"

#include <stdio.h>
void main(void)
 {
    char livro[LINHA];
    char nome_biblio[LINHA];

    printf("O título deste livro é %s\n", TITULO);
    printf(SECAO);
 }
```

Quando você compila um programa C, um pre-processador roda primeiro. O propósito do pré-processador é incluir quaisquer arquivos de cabeçalho especificados e expandir as macros e as constantes. Antes que o compilador C realmente comece a compilar seu programa, o pré-processador substituirá cada nome de constante pelo valor da constante, como mostrado aqui:

```
void main(void)
 {
   char livro[128];
   char nome_biblio[128];
   printf("O nome deste livro é %s\n", "Bíblia do Programador C/C++, do
Jamsa!");
   printf("Macros");
 }
```

Como o pré-processador trabalha com *#define*, *#include* e outros comandos #, esses comandos são normalmente conhecidos como *diretivas do pré-processador*.

NOMEANDO AS CONSTANTES E AS MACROS 134

Como você aprendeu, uma constante é um nome que o compilador associa com um valor que não muda. Na Dica 144, você aprenderá sobre as macros de C. Ao usar constantes e macros dentro de seus programas, você deverá usar nomes representativos que descrevam exatamente o uso delas. Para ajudar os programadores que lerem seu código a diferenciar entre constantes e variáveis, você normalmente deverá usar letras maiusculas para os nomes de constantes e de macros. As seguintes diretivas *#define* ilustram várias definições de macros.

```
#define VERDADEIRO 1
#define FALSO 0
#define PI 3.1415
#define PROGRAMADOR "Kris Jamsa"
```

Como você pode ver, as constantes podem conter valores *int*, *float* e até caracteres *char*.

USANDO A CONSTANTE DE PRÉ-PROCESSADOR _FILE_ 135

Quando você trabalha em um grande projeto, algumas vezes pode querer que o pré-processador saiba o nome do arquivo-fonte atual. Por exemplo, você poderia usar o nome de arquivo dentro de uma diretiva de pré-processador que inclua uma mensagem para o usuario dizendo que o programa ainda está em desenvolvimento, como mostrado aqui:

```
O programa Folha.C ainda está em desenvolvimento e teste.
Esta é uma versão Beta.
```

Para ajudar seus programadores a efetuar esse processamento, o pré-processador C define a constante *__FILE__* como igual ao nome do arquivo-fonte atual. O programa a seguir, *consfile.c*, ilustra o uso da constante *__FILE__*.

```
#include <stdio.h>

void main(void)
 {
   printf("O arquivo %s está em teste Beta\n", __FILE__);
 }
```

Quando você compilar e executar o programa *consfile.c*, sua tela exibirá o seguinte:

```
O arquivo consfile.c está sob teste Beta
C:\>
```

Nota: Embora muitas constantes de pré-processador mudem de um compilador para outro, a constante __FILE__ é consistente dentro do Turbo C++ Lite, do Visual C++, do Borland C++ 5.02 e do Borland C++ Builder.

USANDO A CONSTANTE DE PRÉ-PROCESSADOR __LINE__ 136

Ao trabalhar em um projeto grande, algumas vezes você poderá querer que o pré-processador saiba, e potencialmente use, o número da linha atual do arquivo-fonte atual. Por exemplo, se estiver depurando um programa, você poderia querer que o compilador exibisse mensagens a partir de vários pontos dentro do programa, como mostrado aqui:

```
Cheguei com sucesso à linha 10
Cheguei com sucesso à linha 301
Cheguei com sucesso à linha 213
```

O programa a seguir, *conslin.c*, ilustra o uso da constante de pré-processador *__LINE__*:

```
#include <stdio.h>

void main(void)
```

```
{
  printf("Cheguei com sucesso à linha %d\n", __LINE__);

  // Outros comandos aqui

  printf("Cheguei com sucesso à linha %d\n", __LINE__);
}
```

Quando você compilar e executar o programa, sua tela exibirá o seguinte:

```
Cheguei com sucesso à linha 5
Cheguei com sucesso à linha 7
```

Nota: Embora muitas constantes de pré-processador mudem de um compilador para outro, a constante __LINE__ é consistente com o Turbo C++ Lite, o Visual C++, o Borland C++ 5.02 e o Borland C++ Builder

137 ALTERANDO A CONTAGEM DE LINHA DO PRÉ-PROCESSADOR

Na Dica 136 você aprendeu como usar a constante *__LINE__* do pré-processador dentro de seus programas. Quando você usar a constante *__LINE__*, algumas vezes poderá querer alterar o número da linha atual do pré-processador. Por exemplo, assuma que você esteja usando *__LINE__* para ajudar a depurar seus programas, como discutido na Dica 136. Se você restringiu o erro a um conjunto específico de instruções, pode querer que o pré-processador exiba os números de linha relativos a uma localização específica. Para lhe ajudar a executar esse processamento, o pré-processador C fornece a diretiva *#line* que lhe permite alterar o número da linha atual. A diretiva a seguir, por exemplo, instrui o pré processador a definir seu número de linha para 100:

```
#line 100
```

Você também pode usar a diretiva *#line* para alterar o nome do arquivo de código-fonte que a constante *__FILE__* exibirá:

```
#line 1 "NOMEARQ.C"
```

O programa a seguir, *muda_lin.c*, ilustra como usar a diretiva *#line*.

```
#include <stdio.h>

void main(void)
 {
   printf(
    "Arquivo %s: Cheguei com sucesso à linha %d\n", __FILE__, __LINE__);

   // Outros comandos aqui

#line 100 "NOMEARQ.C"

   printf("Arquivo %s: Cheguei com sucesso à linha %d\n",
      __FILE__, __LINE__);
 }
```

Quando você compilar e executar o programa *muda_lin.c*, sua tela exibirá o seguinte:

```
Arquivo muda_lin.C: Cheguei com sucesso à linha 6
Arquivo NOMEARQ.C: Cheguei com sucesso à linha 102
C:\>
```

138 GERANDO UM ERRO INCONDICIONAL DO PRÉ-PROCESSADOR

À medida que seus programas vão se tornando mais complexos e passarem a usar um número grande de arquivos de cabeçalho, será possível que algumas vezes você não queira que o programa compile com sucesso caso ele não tenha definido uma ou mais constantes. Da mesma forma, se você estiver trabalhando com um grupo de programadores e quiser que eles saibam de uma modificação que você tenha feito no programa, poderá usar a di-

retiva de pré-processador *#error* para exibir uma mensagem de erro e finalizar a compilação. Por exemplo, a diretiva a seguir finaliza a compilação, exibindo uma mensagem para o usuário sobre a atualização:

```
#error A rotina string_classif agora usa strings far
```

Antes que outros programadores possam compilar com sucesso o programa, eles precisam remover a diretiva *#error*, tornando-se, portanto, cientes da alteração que você fez.

OUTRAS CONSTANTES DO PRÉ-PROCESSADOR 139

Várias dicas nesta seção apresentaram constantes do pré-processador que a maioria dos compiladores suporta. Alguns compiladores definem muitas outras constantes de pré-processador. O compilador Microsoft *Visual C++*, por exemplo, usa mais de 15 outras constantes de pré-processador não discutidas neste livro. Consulte a documentação que acompanha seu compilador para determinar se seus programas poderão utilizar outras constantes de pré-processador. Adicionalmente, consulte na documentação on-line o título *Predefined Macros*.

REGISTRANDO A DATA E A HORA DO PRÉ-PROCESSADOR 140

Ao trabalhar em programas grandes, você poderá querer que seu pré-processador trabalhe com a data e a hora atuais. Por exemplo, você pode querer que o programa exiba uma mensagem que informa a data e a hora em que você compilou pela última vez o programa, como mostrado aqui:

```
Teste Beta: FOLHA.C: Última compilação em Jul 8 1998 14:45
```

Para lhe ajudar a realizar esse processamento, o pré-processador C atribui a data e a hora atuais às constantes *__DATE__* e *__TIME__*. O programa a seguir, *datahora.c*, ilustra como você poderia usar as constantes *_DATE_* e *_TIME_*:

```
#include <stdio.h>

void main(void)
 {
   printf("Teste Beta: Última compilação em %s %s\n", __DATE__, __TIME__);
 }
```

TESTANDO A ADESÃO AO PADRÃO ANSI 141

Embora os compiladores C sejam muito similares, cada um deles oferece capacidades exclusivas. Para lhe ajudar a escrever programas que depois você possa migrar facilmente de um sistema para outro, o Instituto Nacional Americano de Normas (American National Standards Institute (ANSI)) define padrões para os operadores, construções, comandos e funções que um compilador deve suportar. Os compiladores que aderem a essas normas são chamados *compiladores ANSI C*. À medida que você cria programas, algumas vezes pode querer determinar se esta ou não usando um compilador ANSI. Para lhe ajudar a fazer isso, os compiladores ANSI C definem a constante *_STDC__* (de STandarD C) Se o compilador estiver configurado para aderir ao padrão ANSI C, ele definirá a constante. Caso contrário, ele não definirá a constante. O programa a seguir, *chc_ansi.c*, usa a constante *STDC__* para determinar se o compilador atual compila para os padrões ANSI:

```
#include <stdio.h>

void main(void)
 {
   #ifdef __STDC__
      printf("Adesão ao ANSI C\n");
   #else
      printf("Não está no modo ANSI C\n");
   #endif
 }
```

Nota: A maioria dos compiladores oferece opções de linha de comando ou pragmas in-line para que você o instrua a aderir ao padrão ANSI. Você aprenderá sobre as opções da linha de comando e pragmas posteriormente, neste livro.

142 TESTANDO C++ VERSUS C

Você pode usar várias das dicas apresentadas neste livro tanto na programação em C quanto em C++, enquanto outras dicas aplicam-se apenas a C++. À medida que você criar seus próprios programas, algumas vezes desejará que o pré-processador determine se você está usando C ou C++ e processe seus comandos apropriadamente. Para lhe ajudar a realizar esses testes, muitos compiladores C++ definem a constante *__cplusplus*. Se você usar um compilador C padrão, a constante estará indefinida. O seguinte programa, *chc_cpp.c*, usa a constante *__cplusplus* para determinar o modo correto do compilador:

```
#include <stdio.h>

void main(void)
 {
   #ifdef __cplusplus
      printf("Usando C++\n");
   #else
      printf("Usando C\n");
   #endif
 }
```

Se você examinar os arquivos de cabeçalho que o compilador fornece, encontrará muitos usos da constante *__cplusplus*.

Nota: Muitos compiladores C++ permitem o uso de opções de linha de comando para instruí-los a compilar usando C++, em vez de C padrão.

143 ANULANDO A DEFINIÇÃO DE UMA MACRO OU CONSTANTE

Várias dicas apresentadas nesta seção discutiram constantes e macros que o pré-processador define ou que o arquivo de cabeçalho contém. Dependendo do seu programa, você pode querer que o pré-processador remova a definição de uma ou mais dessas constantes. Por exemplo, a macro a seguir redefine a macro *_toupper*, que está definida no arquivo de cabeçalho *ctype.h*:

```
#define _toupper(c) ((((c) >= 'a') && ((c) <= 'z')) ? (c) - 'a' + 'A': c)
```

Quando você compilar este programa, muitos pré-processadores exibirão uma mensagem de advertência dizendo que você redefiniu a macro. Para evitar o aparecimento dessa mensagem de advertência, use a diretiva *#undef* para remover a definição atual da macro antes de redefini-la, como mostrado aqui:

```
#undef _toupper
#define _toupper(c) ((((c) >= 'a') && ((c) <= 'z')) ? (c) - 'a' + 'A': c)
```

144 COMPARANDO MACROS E FUNÇÕES

Os novos programadores normalmente ficam confusos sobre quando devem usar macros ou funções por causa das similaridades entre elas. Como você aprendeu, cada vez que o pré-processador encontra uma referência de macro dentro de seu programa, ele substitui a referência pelos comandos da macro. Portanto, se seu programa usa uma macro 15 vezes, ele terá 15 cópias diferentes da macro colocada em seu código. Consequentemente, o tamanho do programa executável crescerá. Por outro lado, quando o programa usa uma função, ele somente contém uma cópia do código, o que reduz o tamanho do programa. Quando o programa usa a função, ele chama (desvia-se para) o código da função. A desvantagem de usar funções, no entanto, é que cada chamada de função incorre em processamento adicional que torna a chamada da função ligeiramente mais demorada de executar que uma macro comparável. Portanto, se você quiser desempenho mais alto, use uma macro. Porém, se o tamanho do programa é o que lhe preocupa mais, use uma função.

COMPREENDENDO PRAGMAS DO COMPILADOR 145

Várias dicas nesta seção apresentaram diferentes diretivas de pré-processador, tais como *#define*, *#include* e *#undef*. Dependendo do seu compilador, seu pré-processador pode suportar várias diretivas de compilador, chamadas *pragmas*. O formato de uma pragma é como segue:

```
#pragma diretiva_compilador
```

Por exemplo, o compilador *Turbo C++ Lite* oferece as pragmas *startup* e *exit*, que lhe permitem especificar as funções que você quer que seu programa execute automaticamente quando o programa iniciar ou terminar:

```
#pragma startup carrega_data
#pragma exit fecha_todos_arquivos
```

Observe que a função que você nomeia dentro da pragma *startup* na verdade será executada antes de *main*, de modo que você não deverá usar a pragma *startup* com muita frequência. Dependendo do seu compilador, as pragmas disponíveis diferirão. Consulte na documentação que acompanha seu compilador uma descrição completa das pragmas disponíveis para seu programa usar.

Nota: Quando você usa as pragmas startup e exit, a função que você nomeia dentro de pragma não pode receber parâmetros e não pode retornar nenhum valor; em outras palavras, você precisa escrever a função da seguinte maneira

```
void function(void)
```

APRENDENDO SOBRE OS VALORES E MACROS PREDEFINIDOS 146

Muitas dicas nesta seção discutiram as macros, as constantes e as várias diretivas de pré-processador. Um dos modos mais eficientes de aprender como usar macros, constantes e outras diretivas de pré-processador é examinar como o compilador C usa essas opções. O compilador C coloca as macros e as constantes dentro de arquivos de cabeçalho que residem no subdiretório *include* do compilador. Muitos dos arquivos de cabeçalho apresentam modos de usar várias diretivas de pré-processador. Examinando o conteúdo de vários arquivos de cabeçalho, você poderá aprender muitos modos de melhorar seus programas utilizando as capacidades do pré-processador.

CRIANDO SEUS PRÓPRIOS ARQUIVOS DE CABEÇALHO 147

Como você sabe, o compilador C fornece diferentes arquivos de cabeçalho que contêm macros, constantes e protótipos de funções relacionados. À medida que aumentar o numero de programas que você criar, será possíve. descobrir que muitos dos seus programas usam as mesmas constantes e macros. Em vez de digitar repetidamente essas macros e constantes em seus programas, você poderá considerar criar seu próprio arquivo de cabeçalho e colocar no arquivo as macros e constantes correspondentes. Assumindo que você crie um arquivo de cabeçalho chamado *min_defs.h*, poderá incluir o arquivo no início de seu programa usando a diretiva *#include* do pré-processador, como mostrado aqui:

```
#include "min_defs.h"
```

Quando você incluir suas macros e constantes em um arquivo de cabeçalho desse modo, poderá rapidamente modificar vários programas editando o arquivo de cabeçalho, e, depois, recompilando os programas que incluem o arquivo.

USANDO #INCLUDE <NOMEARQ.H> OU #INCLUDE "NOMEARQ.H" 148

Todos os programas apresentados neste livro incluíram o arquivo de cabeçalho *stdio.h*, como mostrado aqui:

```
#include <stdio.h>
```

Na Dica 147 você aprendeu como criar e incluir seu próprio arquivo de cabeçalho, *min_defs.h*. Você pode incluir *stdio.h* e *min_defs.h* dentro de seus programas com os seguintes comandos:

```
#include <stdio.h>
#include "min_defs.h"
```

Observe nos dois comandos *include* que os sinais de menor e maior (<>) delimitam o arquivo de cabeçalho *stdio.h*, enquanto *min_defs.c* aparece delimitado por aspas. Quando você delimitar o nome do arquivo de cabeçalho com os sinais de menor e maior, o compilador C procurará o arquivo especificado no seu diretório de arquivos de cabeçalho. Se o arquivo for encontrado, será usado pelo pré-processador. Se o compilador não localizar o arquivo, ele o procurará no diretório atual ou em um diretório que você especificar. Por outro lado, quando você delimitar o nome de arquivo de cabeçalho por aspas, o compilador somente procurará o arquivo no diretório atual

149 TESTANDO SE UM SÍMBOLO ESTÁ DEFINIDO

Várias dicas nesta seção apresentaram símbolos predefinidos do compilador C. Além disso, algumas dicas discutiram como você pode definir suas próprias constantes e macros. Dependendo do seu programa, é possível que algumas vezes você queira que o pré-processador teste se o programa definiu um símbolo anteriormente e, em caso afirmativo, processe um determinado conjunto de comandos. Para ajudar seu programa a testar se o programa definiu anteriormente um símbolo, o pré-processador C suporta a diretiva *#ifdef*. O formato de *#ifdef* é como segue:

```
#ifdef símbolo
   // comandos
#endif
```

Quando o pré-processador encontra a diretiva *#ifdef*, ele testa se o programa definiu anteriormente o símbolo especificado. Em caso afirmativo, ele processa os comandos que seguem a diretiva até encontrar o comando *#endif*. Às vezes você quer que o programa processe comandos caso o programa não tenha definido um símbolo. Em tais casos, pode usar a diretiva *#ifndef*. Os comandos a seguir usam *#ifndef* para instruir o pré-processador a definir a macro *_toupper* se uma macro similar não estiver definida:

```
#ifndef _toupper
   #define _toupper(c) ((((c) >= 'a') && ((c) <= 'z')) ? (c) - 'a' + 'A' : c)
#endif
```

150 EFETUANDO O PROCESSAMENTO IF-ELSE

Na Dica 149 vimos como usar os comandos *#ifdef*, *#ifndef* e *#endif* para especificar um conjunto de comandos que você quer que o processador execute se um programa anteriormente tiver definido (*#ifdef*) ou não definido (*#ifndef*) um símbolo. Algumas vezes você poderá querer levar esse processamento um passo adiante e incluir um conjunto de comandos que deseja que o processador execute quando a condição testada dentro do comando *#ifdef* for verdadeira, e um conjunto diferente se a condição for falsa. Para executar esse processamento, você poderá usar a diretiva *#else*, mostrada aqui:

```
#ifdef símbolo
    // Comandos
#else
    // Outros comandos
#endif
```

Por exemplo, ambos os compiladores Microsoft *Visual C++* e o *Borland C++ 5.02* incluem constantes de pré-processador exclusivas que indicam qual compilador e qual versão você está usando para compilar o programa. Você pode usar as constantes exclusivas de cada compilador para responder a diferentes compiladores. Por exemplo, o fragmento de código a seguir imprimirá *Microsoft* se o compilador for o *Visual C++* e *Borland* se o compilador for o *Borland C++ 5.02*.

```
#ifdef MSC_VER
   printf ("Microsoft");
#endif
#ifdef __BORLANDC__
```

```
    printf ("Borland");
#endif
```

EFETUANDO UM TESTE DE CONDIÇÃO DE PRÉ-PROCESSADOR MAIS PODEROSO 151

Na dica 149 você aprendeu como usar os comandos *#ifdef* e *#ifndef* para instruir o pré-processador a testar anteriormente se um programa definiu ou nao um símbolo, e, depois, processar os comandos que seguem com base no resultado do teste. Em muitos casos, você poderá precisar que o processador teste se vários símbolos estão definidos, nao definidos, ou alguma combinaçao de ambas as coisas. As diretivas a seguir testam primeiro se o símbolo *MINHA_BIBLIO* está definido. Se seu programa definiu anteriormente *MINHA_BIBLIO*, as diretivas do pré-processador testam se seu programa também definiu anteriormente o símbolo *MINHAS_FUNC*. Se seu programa ainda nao definiu *MINHAS_FUNC*, o cód.go instrui o pré-processador a incluir o arquivo de cabeçalho *meu_cdg.h*:

```
#ifdef MINHA_BIBLIO
#ifndef MINHAS_ROTINAS
#include "meu_cdg.h"
#endif
#endif
```

Embora as diretivas efetuem o processamento desejado, as condicionais embutidas uma dentro de outra o tornam potencialmente difícil para outro programador entender o que você deseja. Como uma alternativa, seus programas podem usar a diretiva *#if* com o operador *defined* para testar se o programa definiu anteriormente o símbolo, como mostrado aqui:

```
#if defined (símbolo)
    // comandos
#endif
```

Sua vantagem em usar o comando *#if* é que você pode combinar teste, ao contrário do que acontece com as diretivas *#ifdef* e *#ifndef*. A diretiva a seguir executa o mesmo teste do primeiro exemplo nesta dica:

```
#if defined(MINHA_BIBLIO)  && !defined(MINHAS_ROTINAS)
#include "meu_cdg.h"
#endif
```

Você poderá usar *#if defined* para criar condições que usam operadores lógicos de C (incluindo &&, ||, e !).

REALIZANDO PROCESSAMENTO IF-ELSE E ELSE-IF 152

Como visto na dica anterior, é fácil usar a diretiva *#if* do pré-processador para testar se seu programa definiu ou não um símbolo. Ao usar a diretiva *#if*, você algumas vezes poderá querer que o pré-processador processe um conjunto de comandos, quando um símbolo estiver definido, e outro conjunto caso o símbolo não esteja definido (*pré-processamento condicional*). Você poderá realizar pré-processamento condicional usando a diretiva *#else*:

```
#if defined(símbolo)
    // Comandos
#else
    // Comandos
#endif
```

Levando o exempo de pré-processamento anterior um passo mais longe, algumas vezes você poderá querer que o pré-processador teste o status de outros símbolos quando uma condição especificada não passar. As diretivas a seguir, por exemplo, instruem o pré-processador a processar um conjunto de comandos se o símbolo *MINHA_BIBLIO* estiver definido; outro conjunto se *MINHA_BIBLIO* não estiver definido e caso *MINHAS_ROTINAS* esteja definido, e um terceiro conjunto se nenhum dos símbolos estiver definido:

```
#if defined (MINHA_BIBLIO)
```

```
    // Comandos
#else if defined (MINHAS_ROTINAS)
    // Comandos
#else
    // Comandos
#endif
```

Como você pode ver, ao usar as diretivas *#if* e *#else*, seu controle sobre o pré-processador aumenta significativamente.

Nota: Alguns compiladores, incluindo o Turbo C++ Lite, suportam a diretiva de pré-processador #elif, que efetua o mesmo processamento que a construção #else if.

153 Definindo Macros e Constantes Que Requerem Múltiplas Linhas

Várias dicas apresentadas nesta seção definiram constantes e macros. À medida que suas constantes e macros vão se tornando mais complexas, algumas vezes uma definição não caberá em uma única linha. Quando você precisar quebrar a definição de uma constante ou de uma macro para a próxima linha, coloque um caractere de barra invertida no final da linha, como mostrado aqui:

```
#define string_caracteres muito_longa "Esta constante \
string extremamente longa requer duas linhas"

#define _toupper(c) ((((c) >= 'a') && ((c) <= 'z'))  \
   ? (c)   'a' + 'A': c)
```

154 Criando Suas Próprias Macros

Como você aprendeu, as macros oferecem um modo de definir constantes que o pré-processador substitui em todo o seu programa antes que a compilação inicie. Além disso, as macros lhe permitem criar operações similares às funções que trabalham como *parâmetros*. Os parâmetros são valores que você passa para a macro. Por exemplo, a seguinte macro, *SOMA*, retorna a soma dos dois valores que você passa para a macro:

```
#define SOMA(x, y) ((x) + (y))
```

O programa a seguir, *ex_soma.c*, usa a macro *SOMA* para somar vários valores:

```
#include <stdio.h>

#define SOMA(x, y) ((x) + (y))

void main(void)
 {
   printf("Somando 3 + 5 = %d\n", SOMA(3, 5));
   printf("Somando 3.4 + 3.1 = %f\n", SOMA(3.4, 3.1));
   printf("Somando -100 + 1000 = %d\n", SOMA(-100, 1000));
 }
```

Dentro da definição da macro *SOMA*, *x* e *y* representam os parâmetros da macro. Quando você passa dois valores para a macro, tal como *SOMA(3, 5)* o pré-processador substitui os parâmetros dentro da macro, como mostrado na Figura 154.

No programa *ex_soma.c*, as substituições do pré-processador resultarão no seguinte código:

```
printf("Somando 3 + 5 = %d\n", ((3) + (5)));
printf("Somando 3.4 + 3.1 = %f\n", ((3.4) + (3.1)));
printf("Somando -100 + 1000 = %d\n", ((-100) + (1000)));
```

Figura 154 A substituição de parâmetro para *SOMA*.

NÃO COLOQUE PONTO-E-VÍRGULA NAS DEFINIÇÕES DE MACROS 155

Quando você examinar a definição de macro da seguinte macro *SOMA*, observe que a macro não inclui um ponto-e-vírgula:

```
#define SOMA(x, y) ((x) + (y))
```

Se você incluir um ponto-e-vírgula dentro de sua macro, o pré-processador colocará o ponto-e-vírgula em cada ocorrência de macro em todo o seu programa. Por exemplo, assuma que você tenha colocado um ponto-e-vírgula no final da definição da macro *SOMA*, como mostrado aqui:

```
#define SOMA(x, y) ((x) + (y));
```

Quando o pré-processador expandir a macro, ele incluirá o ponto-e-vírgula, como mostrado aqui:

```
printf("Somando 3 + 5 = %d\n", ((3) + (5))););
printf("Somando 3.4 + 3.1 = %f\n", ((3.4) + (3.1))););
printf("Somando -100 + 1000 = %d\n", ((-100) + (1000))););
```

Como o ponto-e-vírgula agora ocorre no meio do comando *printf* (indicando o final do comando), o compilador gerará erros.

Nota: A não ser que você queira que o pré-processador inclua um ponto-e- vírgula na expansão da macro, não inclua um ponto-e-vírgula na definição da macro.

CRIANDO MACROS MIN E MAX 156

Na Dica 154 você criou a macro *SOMA*, que somou dois valores juntos. As macros seguintes, *MIN* e *MAX*, retornam o mínimo e o máximo de dois valores:

```
#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))
```

O programa a seguir, *min_max.c*, ilustra como usar as macros *MIN* e *MAX*:

```
#include <stdio.h>

#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))

void main(void)
 {
   printf("O maior entre 10.0 e 25.0 é %f\n", MAX(10.0, 25.0));
   printf("O menor entre 3.4 e 3.1 é %f\n", MIN(3.4, 3.1));
 }
```

Quando você executar o programa *min_max.c*, as substituições do pré-processador resultarão no seguinte código:

```
printf("O maior entre 10.0 e 25.0 é %f\n", (((10.0) < (25.0)) ? (10.0): (25.0)));
printf("O menor entre 3.4 e 3.1 é %f\n", (((3.4) > (3.1)) ? (3.4): (3.1)));
```

157 CRIANDO MACROS QUADRADO E CUBO

Você viu que, C lhe permite definir e passar valores para as macros. As duas últimas macros que você examinará nesta seção são *QUADRADO* e *CUBO*, que retornam, respectivamente, um valor ao quadrado (x * x) e um valor ao cubo (x * x * x):

```
#define QUADRADO(x) ((x) * (x))
#define CUBO(x) ((x) * (x) * (x))
```

O programa a seguir, *quad_cubo.c*, ilustra como usar as macros *QUADRADO* e *CUBO*:

```
#include <stdio.h>
#define QUADRADO(x) ((x) * (x))
#define CUBO(x) ((x) * (x) * (x))

void main(void)
 {
   printf("O quadrado de 2 é %d\n", quadrado (2));
   printf("O cubo de 100 é %f\n", cubo (100.0));
 }
```

No programa *quad_cubo.c*, as substituições do pré-processador resultam no seguinte código:

```
   printf("O quadrado de 2 é %d\n", ((2) * (2)));
   printf("O cubo de 100 é %f\n", ((100.0) * (100.0) * (100.0)));
```

*Nota: Para evitar o extravasamento, o programa **quad_cubo.c** usa o valor em ponto flutuante 100.0 dentro da macro CUBO.*

158 CUIDADO COM OS ESPAÇOS NAS DEFINIÇÕES DAS MACROS

Várias dicas anteriores apresentaram macros que recebem parâmetros. Quando você define macros que recebem parâmetros, precisa tomar cuidado com espaço em branco na definição de macro. Não coloque um espaço entre o nome da macro e seus parâmetros. Por exemplo, considere a seguinte definição de macro, *QUADRADO*.

```
#define QUADRADO (x) ((x) * (x))
```

Quando o pré-processador examina seu programa, os espaços em branco entre o nome da macro fazem o pré-processador assumir que deve substituir cada ocorrência do nome *QUADRADO* com *(x) ((x) * (x))*, em vez de substituir com *((x) * (x))*. Como resultado, a macro não avalie corretamente, e, na maioria dos casos, o compilador gerará mensagens de erro de sintaxe ou advertências por causa da substituição. Para compreender melhor o processo de substituição da macro do pré-processador, experimente o programa *quad_cubo.c* (apresentado na Dica 157), colocando um espaço após cada nome de macro.

159 COMPREENDENDO COMO USAR OS PARÊNTESES

Em muitas dicas anteriores vimos macros às quais seus programas passarão valores (parâmetros). Se você der uma olhada nas definições de cada macro, verá que os valores estão envolvidos por parênteses:

```
#define SOMA(x, y) ((x) + (y))
#define QUADRADO(x) ((x) * (x))
#define CUBO(x) ((x) * (x) * (x))
```

```
#define MIN(x, y) (((x) < (y)) ? (x): (y))
#define MAX(x, y) (((x) > (y)) ? (x): (y))
```

As definições de macros colocam os parâmetros dentro de parênteses para suportar as expressões. Como um exemplo, considere o seguinte comando:

```
result = QUADRADO(3 + 5);
```

O comando deve atribuir à variável *result* o valor 64 (8 * 8). Por exemplo, assuma que você venha a definir a macro *QUADRADO*, como segue:

```
#define QUADRADO(x) (x * x)
```

Quando o pré-processador substitui a expressão 3 + 5 para x, a substituição torna-se a seguinte:

```
result = (3 + 5 * 3 + 5);
```

Lembre-se da precedência de operadores de C e observe que a multiplicação tem precedência mais alta que a adição. Portanto, o programa calcularia a expressão como segue:

```
result = (3 + 5 * 3 + 5);
       = ( 3 + 15 + 5);
       = 23;
```

No entanto, quando você coloca cada parâmetro dentro de parênteses, garante que o pré-processador avaliará a expressão corretamente:

```
result = quadrado (3+5);
       = ((3 + 5) * (3 + 5));
       = ((8) * (8));
       = (64);
       = 64;
```

Nota: Como regra, você sempre deverá colocar os parâmetros de suas macros dentro de parênteses.

AS MACROS NÃO TÊM TIPO 160

Na seção Funções, mais à frente você aprenderá como criar funções que efetuam operações específicas. Você aprenderá que C lhe permite passar parâmetros para suas funções, exatamente como você passou valores para as macros. Se sua função executa uma operação e retorna um resultado, você precisará especificar o tipo do resultado (que é *int*, *float*, e assim por diante). Por exemplo, a seguinte função, *soma_valores*, soma dois valores inteiros e retorna um resultado do tipo *int*:

```
int soma_valores(int x, int y)
 {
    return (x + y);
 }
```

Dentro de seu programa você pode somente usar a função *soma_valores* para somar dois valores do tipo *int*. Se você tentar somar dois valores em ponto flutuante, um erro ocorrerá. Como você viu, as macros lhe permitem trabalhar com valores de qualquer tipo. Por exemplo, a macro *SOMA*, que você criou anteriormente, suportava valores dos tipos *int* e *float*:

```
printf("Somando 3 + 5 = %d\n", SOMA(3, 5));
printf("Somando 3.4 + 3.1 = %f\n', SOMA(3.4, 3.1));
```

Quando você usa macros para operações aritméticas simples, elimina a necessidade de funções duplicadas simplesmente porque quer trabalhar com valores de tipos diferentes. No entanto, como você aprenderá posteriormente na seção Funções, existem outras vantagens e desvantagens a considerar ao decidir se você vai usar macros ou funções.

161 VISUALIZANDO UMA STRING DE C

Seu computador requer um byte de memória para armazenar um único caractere ASCII. Como você aprendeu, uma string é uma seqüência de caracteres ASCII. Quando você declara uma constante de string, C automaticamente atribui o caractere *NULL*. Quando seus programas criam suas próprias strings lendo caracteres do teclado, eles precisam colocar o caractere *NULL* no final da string para indicar onde ela termina. Portanto, o melhor modo para você visualizar uma string é como uma coleção de bytes terminados por um caractere *NULL*, como mostrado na Figura 161.

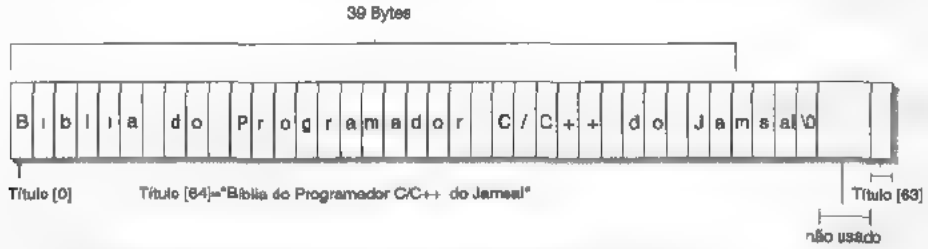

Figura 161 C armazena as strings em posições de byte consecutivos na memória.

Quando as funções trabalham com strings, em geral a função somente recebe a localização onde a string inicia. Após a função saber a localização inicial da string, ela pode percorrer posições sucessivas de memória até que a função encontre o caractere *NULL* (que indica o final da string).

162 COMO O COMPILADOR REPRESENTA UMA STRING DE CARACTERES

Várias dicas apresentadas neste livro usam constantes de string de caracteres delimitadas por aspas, como no exemplo a seguir:

```
"Bíblia do Programador C/C++, do Jamsa!"
```

Quando você usa uma constante de string de caracteres dentro de seu programa, o compilador C automaticamente coloca o caractere *NULL* (\0) no final da string. Dada a constante string anterior, o compilador C armazenará a constante na memória, como mostrado na Figura 162.

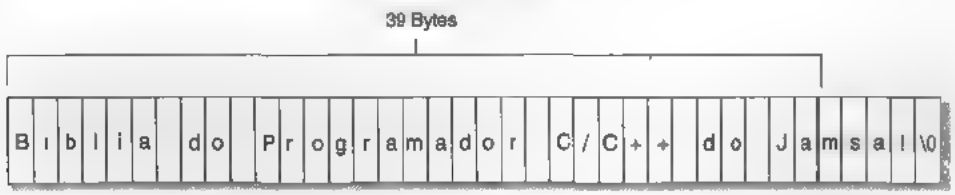

Figura 162 C automaticamente coloca o caractere NULL nas constantes de string.

163 COMO C ARMAZENA AS STRINGS DE CARACTERES

Muitas das dicas que este livro apresenta utilizam strings de caracteres. Por exemplo, alguns programas usam strings para ler arquivos e entrada do teclado e efetuar outras operações. Em C, uma string de caracteres é uma matriz de caracteres terminada por zero ou *NULL*. Para criar uma string de caracteres, você simplesmente declara uma matriz de caracteres, como mostrado aqui:

```
char string[256];
```

C criará uma string capaz de armazenar 256 caracteres, que C indexa de *string[0]* até *string[255]*. Como a string pode conter menos que 256 caracteres, C usa o caractere *NULL* (ASCII 0) para representar o último caractere da string. C tipicamente não coloca o caractere *NULL* após o último caractere na string. Em vez disso,

funções tais como *fgets* ou *gets* colocam o caractere *NULL* no final da string. À medida que seus programas manipularem strings, é sua responsabilidade garantir que o caractere *NULL* esteja presente. O programa a seguir, *cria_abc.c*, define uma string de 256 caracteres, e, depois, atribui as letras maiusculas do alfabeto aos primeiros vinte e seis dígitos da string:

```
#include <stdio.h>

void main(void)
 {
   char string[256];
   int i;

   for (i = 0; i < 26; i++)
      string[i] = 'A' + i;
   string[i] = NULL;
   printf ("A string contém %s\n', string);
 }
```

O programa *cria_abc.c* usa o laço *for* para atribuir as letras A até Z à string. O programa depois coloca o caractere *NULL* após a letra Z para indicar o final da string. A função *printf* então exibirá cada caractere na string até o caractere *NULL*. As funções que trabalham com strings usam o caractere *NULL* para determinar o final da string. O programa a seguir, *a_ate_z.c*, também atribui as letras A até Z a uma string de caracteres. No entanto, depois, o programa atribui o caractere *NULL* à *string[10]*, que é a posição que segue imediatamente a letra J. Quando *printf* exibe o conteúdo da string, ele parará na letra J:

```
#include <stdio.h>

void main(void)
 {
   char string[256];
   int i;

   for (i = 0; i < 26; i++)
     string[i] = 'A' + i;
   string[10] = NULL;
   printf ("A string contém %s\n', string);
 }
```

Nota: Quando você trabalhar com strings, sempre precisará incluir o caractere NULL para representar o final da string.

APRENDENDO COMO 'A' DIFERE DE "A" 164

Como você aprendeu na Dica 161, uma string de caracteres é uma seqüência de zero ou mais caracteres ASCII que C tipicamente finaliza com *NULL* (o ASCII 0). Quando você trabalha com caracteres dentro de C, pode usar o valor ASCII numérico de um caractere ou pode colocar o caractere dentro de apóstrofes, tal como 'A'. Por outro lado, quando você usa aspas, tal como "A", C cria uma string de caracteres que contém a letra especificada (ou letras), e finaliza a string com o caractere *NULL*. A Figura 164 ilustra como C armazena as constantes 'A' e "A".

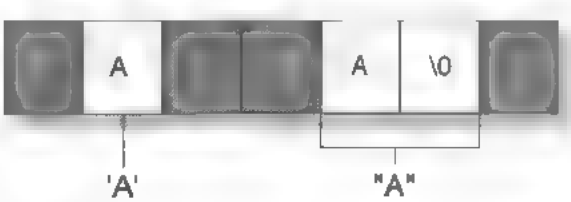

Figura 164 Como C armazena as constantes 'A' e "A".

Como C as armazena de forma diferente, as constantes de caractere e string não são iguais, e você precisará tratá-las de forma diferente dentro de seus programas.

165 REPRESENTANDO UM APÓSTROFO DENTRO DE UMA CONSTANTE DE STRING

Como você aprendeu, para criar uma constante de string, seu programa precisará colocar os caracteres desejados dentro de aspas:

```
"Isto é uma constante de string"
```

Dependendo dos seus programas, algumas vezes uma constante de string conterá um caractere aspas. Por exemplo, assuma que você queira representar a seguinte string:

```
"Pare!", ele disse.
```

Como C usa as aspas para definir as constantes de string, você precisará ter um modo de dizer ao compilador que quer incluir aspas dentro da string. Para fazer isso, use a seqüência de escape \", como mostrado aqui:

```
"\"Pare!\", ele disse."
```

O programa a seguir, *aspas.c*, usa a sequência de escape \" para colocar aspas dentro de uma constante de string:

```
#include <stdio.h>

void main(void)
 {
   char string[] = "\"Pare!\", ele disse.";

   printf(string);
 }
```

166 DETERMINANDO O TAMANHO DE UMA STRING

Na dica 163 você aprendeu que as funções de C normalmente usam o caractere *NULL* para representar o final de uma string. Funções tais como *fgets* e *cgets* automaticamente atribuem o caractere *NULL* para indicar o final de uma string. O programa a seguir, *exib_str.c*, usa a função *gets* para ler uma string de caracteres do teclado. O programa depois usa um laço *for* para exibir os caracteres da string um de cada vez até que a condicional *string[i] != NULL* seja avaliada como falsa:

```
#include <stdio.h>

void main(void)
 {
   char string[256];  // String digitada pelo usuário
   int i;             // índice para a string

   printf("Digite uma string de caracteres e pressione Enter:\n");
   gets(string);
   // Exibe cada string de caracteres até que NULL seja encontrado
   for (i = 0; string[i] != NULL; i++)
     putchar(string[i]);
   printf("\nO número de caracteres na string é %d\n", i);
 }
```

167 USANDO A FUNÇÃO STRLEN

À medida que você trabalha com strings dentro de seus programas, efetuará muitas operações com base no número de caracteres na string. Para lhe ajudar a determinar o número de caracteres em uma string, a maioria dos compiladores C fornece uma função *strlen*, que retorna o número de caracteres em uma string. O formato da função *strlen* é como segue:

```
#include <string.h>

size_t strlen(const char string);
```

O programa a seguir, *strlen.c*, ilustra como usar a função *strlen*.

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char titulo_livro[] = "Bíblia do Programador C/C++, do Jamsa!";

   printf("%s contém %d caracteres\n", titulo_livro,
      strlen(titulo_livro));
 }
```

Quando você compilar e executar o programa *strlen.c*, sua tela exibirá o seguinte:

```
Bíblia do Programador C/C++, do Jamsa! contém 38 caracteres
C:\>
```

Para compreender melhor como a função *strlen* funciona, considere a seguinte implementação. A função simplesmente conta os caracteres em uma string até, mas não incluindo, o caractere *NULL*:

```
size_t strlen(const char string)
 {
   int i = 0;
   while (string[i])
      i++;
   return(i);
 }
```

Copiando os Caracteres de uma String em Outra String 168

Quando seus programas trabalharem com strings, algumas vezes você precisará copiar o conteúdo de uma string em outra string. Para lhe ajudar a realizar operações em strings de caracteres, a maioria dos compiladores C oferece uma função *strcpy*, que copia os caracteres em uma string (o parâmetro *origem*) em outra string (o parâmetro *destino*).

```
#include <string.h>

char *strcpy(char *destino, const char *origem);
```

A função *strcpy* retorna um ponteiro para o início da string de destino. O programa a seguir, *strcpy.c*, ilustra como você usará a função *strcpy* dentro de seus programas.

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
  char titulo[]="Bíblia do Programador C/C++, do Jamsa!";
  char livro[128];

  strcpy(livro, titulo);
  printf("Nome do livro %s\n", livro);
 }
```

Para compreender melhor como a função *strcpy* funciona, considere a seguinte implementação

```
char *strcpy(char *destino, const char *origem)
 {
   while (*destino++ = *origem++)
      ;
```

```
    return(destino-1);
  }
```

A função *strcpy* simplesmente copia letras da string de origem para o destino até e incluindo o caractere *NULL*.

169 ANEXANDO O CONTEÚDO DE UMA STRING EM OUTRA

Quando seus programas trabalharem com strings, algumas vezes você precisará anexar o conteúdo de uma string em outra string. Por exemplo, se uma string contém um nome de subdiretório e outra contém um nome de arquivo, você poderá anexar o nome de arquivo ao subdiretório para criar um nome de caminho completo. Os programadores C referenciam o processo de anexar uma string em outra como *concatenação de strings*. Para lhe ajudar a anexar uma string em outra, a maioria dos compiladores C fornece uma função chamada *strcat*, que concatena (anexa) uma string de origem em uma string-alvo como mostrado aqui:

```
#include <string.h>

char *strcat (char alvo, const char *origem);
```

O programa a seguir, *strcat.c*, ilustra como usar a função *strcat*.

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char nome[64] = "Dálmata Macho"; strcat(nome, " Happy");
   printf("O nome completo do Happy é %s\n", nome);
 }
```

Quando você compilar e executar o programa *strcat.c*, sua tela exibirá o seguinte:

```
O nome completo do Happy é Dálmata Macho Happy
C:\>
```

Para compreender melhor a função *strcat*, considere a seguinte implementação:

```
char *strcat(const *alvo, const char *origem)
 {
   char *original = alvo;

   while (*alvo)
      alvo++;      // Encontra o final da string
   while (*alvo++ = *origem++)
      ;
   return (original);
 }
```

Como você pode ver, a função *strcat* percorre os caracteres da string de destino até que a função encontre o caractere *NULL*. A função *strcat*, então, anexa cada caractere na string de origem, até e incluindo o caractere *NULL*, na string de destino.

170 ANEXANDO N CARACTERES EM UMA STRING

Na Dica 169 você aprendeu que a função *strcat* lhe permite anexar (concatenar) os caracteres de uma string em outra. Em alguns casos, você não vai querer anexar todos os caracteres em uma string, mas, em vez disso, somente os dois, três, ou *n* primeiros caracteres na string. Para lhe ajudar a anexar *n* caracteres a uma string, a maioria dos compiladores C fornece uma função chamada *strncat*, que anexa os primeiros *n* caracteres de uma string de origem a uma string de destino, como mostrado aqui:

```
#include <stdlib.h>

char *strncat(char *destino, const *origem, size_t n);
```

Se *n* especificar um número de caracteres maior que o número de caracteres na string *origem*, *strncat* copiará caracteres até o final da string e não mais. O programa a seguir, *strncat.c*, ilustra como usar a função *strncat*.

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char nome[64] = "Fernando";

   strncat(nome, " e Rute", 8);
   printf("Você votou em %s?\n", nome);
 }
```

Quando você compilar e executar o programa *strncat.c*, sua tela exibirá o seguinte:

```
Você votou em Fernando e Rute?
C:\>
```

Para lhe ajudar a entender a função *strncat*, considere a seguinte implementação:

```
char *strncat(char *destino, const char *origem, int n)
 {
    char *original = destino;
   int i = 0;

    while (*destino)
     destino++;
    while ((i++ < n) && (*destino++ = *origem++))
    ;
   if (i > n)
     *destino = NULL;
   return(original);
 }
```

TRANSFORMANDO UMA STRING EM OUTRA 171

Várias dicas neste livro mostraram como copiar o conteúdo de uma string em outra. A função *strxfrm* copia o conteúdo de uma string em outra (até o número de caracteres que você especifica dentro do parâmetro *n*) e, depois, retorna o tamanho da string resultante:

```
#include <string.h>
size_t strxfrm(char *alvo, char *origem, size_t n);
```

O parâmetro *alvo* é um ponteiro para o qual a função *strxfrm* copia a string *origem*. O parâmetro *n* especifica o número máximo de caracteres a copiar. O programa a seguir, *strxfrm.c*, ilustra como usar a função *strxfrm*.

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char buffer[64] = "Bíblia do Programador C/C++, do Jamsa!";
   char alvo[64];
   int tamanho;

   tamanho = strxfrm(alvo, buffer, sizeof(buffer));
   printf("Tamanho %d Alvo %s Buffer %s\n", tamanho,
     alvo, buffer);
 }
```

NÃO ULTRAPASSE O TAMANHO DE UMA STRING 172

Muitas dicas nesta seção apresentaram funções que copiam ou anexam caracteres de uma string em outra. Quando você efetua operações de strings de caracteres, precisa garantir que não sobrescreve as localizações de memória da string. Como um exemplo dos problemas com a sobrescrita dos limites de uma string, considere a seguinte declaração, que cria uma string de caracteres capaz de armazenar 10 caracteres.

```
char string[10];
```

Se você atribuir mais de 10 caracteres à string, seu sistema operacional poderá não detectar o erro. Em vez disso, os caracteres que você queria atribuir à string podem sobrescrever as posições de memória que correspondem às outras variáveis. Corrigir um erro de sobrescrita é muito difícil, e o erro também pode fazer seu programa e o sistema operacional interromper a execução. Como regra, declare suas strings ligeiramente maiores do que você acha que será necessário. Fazendo isso, você reduz a possibilidade de sobrescrever uma string. Se seus programas experimentarem erros intermitentes, examine o código de seu programa para determinar se seu programa pode estar sobrescrevendo uma string de caracteres.

173 DETERMINANDO SE DUAS STRINGS SÃO OU NÃO IGUAIS

Quando seus programas trabalham com strings, você frequentemente comparará duas strings para determinar se elas são ou não iguais. Para lhe ajudar a determinar se duas strings contêm os mesmos caracteres, você poderá usar a função *streql*, mostrada aqui:

```
int streql (char *str1, char *str2)
 {
   while ((*str1 == *str2) && (*str1))
     {
       str1++;
       str2++;
     }
   return((*str1 == NULL) && (*str2 == NULL));
 }
```

A função *streql* retornará o valor 1 se as duas strings forem iguais, e 0 se as strings não forem iguais. O seguinte programa C, *streql.c*, ilustra como usar a função *streql*:

```
#include <stdio.h>

void main(void)
 {
   printf("Testando Abc e Abc %d\n", streql("Abc", "Abc"));
   printf("Testando abc e Abc %d\n", streql("abc", "Abc"));
   printf("Testando abcd e abc %d\n", streql("abcd",
      "abc"));
 }
```

Quando você compilar e executar o programa *streql.c*, sua tela exibirá a seguinte saída:

```
Testando Abc e Abc 1
Testando abc e Abc 0
Testando abcd e abc 0
C:\>
```

174 IGNORANDO A CAIXA AO DETERMINAR SE AS STRINGS SÃO IGUAIS

Na Dica 173 você criou a função *streql*, que permite que seus programas determinem rapidamente se duas strings são iguais. Quando a função *streql* compara duas strings, *streql* considera os caracteres maiúsculos e minúsculos como diferentes. Algumas vezes você pode querer comparar duas strings sem considerar a caixa. Para comparar strings sem considerar a caixa, você pode criar a função *strieql*, como mostrado aqui.

```
#include <ctype.h>

int strieql(char *str1, char *str2)
 {
   while ((toupper(*str1) == toupper(*str2)) && (*str1))
     {
       str1++;
       str2++;
     }
```

```
    return((*str1 == NULL) && (*str2 == NULL));
  }
```

Como você pode ver, a função *strieql* converte cada caractere em cada string para maiúscula antes de comparar as duas strings. O programa a seguir, *strieql.c*, ilustra como usar *strieql*.

```
#include <stdio.h>
#include <ctype.h>

void main(void)
 {
   printf("Testando Abc e Abc %d\n", strieql("Abc", "Abc"));
   printf("Testando abc e Abc %d\n", strieql("abc", "Abc"));
   printf("Testando abcd e abc %d\n", strieql("abcd", "abc"));
 }
```

Quando você compilar e executar o programa *strieql.c*, sua tela exibirá o seguinte:

```
Testando Abc e Abc 1
Testando abc e Abc 1
Testando abcd e abc 0
C:\>
```

CONVERTENDO UMA STRING DE CARACTERES PARA MAIÚSCULAS OU MINÚSCULAS 175

Quando seus programas trabalharem com strings, algumas vezes você desejará converter a string para maiúsculas Por exemplo, quando um usuário digitar o nome de um arquivo ou de um cliente, você poderá querer que o programa converta a string digitada em maiúsculas para simplificar as operações de comparação ou para garantir que o programa armazene dados em um formato consistente. Para lhe ajudar a efetuar essas conversões, a maioria dos compiladores C fornece as funções *strlwr* e *strupr*, como mostrado aqui:

```
#include <string.h>
char *strlwr(char *string);
char *strupr(char (string);
```

O programa a seguir, *strcase.c*, ilustra como usar as funções *strlwr* e *srtupr:*

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
  printf(strlwr ("Bíblia do Programador C/C++, do Jamsa!\n"));
  printf(strupr ("Bíblia do Programador C/C++, do Jamsa!\n"));
 }
```

Para lhe ajudar a compreender melhor essas duas funções, considere a seguinte implementação de *strlwr*.

```
#include <ctype.h>
char *strlwr(char *string)
 {
   char *original = string;
   while (*string)
     {
       *string = tolower(*string);
       string++;
     }
   return(original);
 }
```

Como você pode ver, tanto *strlwr* quanto *strupr* percorrem os caracteres em uma string, convertendo cada caractere para maiúscula ou para minúscula, dependendo da função chamada.

176 OBTENDO A PRIMEIRA OCORRÊNCIA DE UM CARACTERE EM UMA STRING

À medida que seus programas trabalham com strings, algumas vezes você pode querer encontrar a primeira ocorrência (a que está mais à esquerda) de um caractere específico dentro de uma string. Por exemplo, se você estiver trabalhando com uma string que contém um nome de caminho, você poderá procurar na string o primeiro caractere de barra invertida (\). Para lhe ajudar a procurar a primeira ocorrência em uma string, a maioria dos compiladores fornece uma função chamada *strchr*, que retorna um ponteiro para a primeira ocorrência de um caractere específico dentro de uma string, como mostrado aqui:

```
#include <string.h>

char *strchr(const char *string, int caractere);
```

Se *strchr* não encontrar o caractere especificado dentro da string, *strchr* retornará um ponteiro para o caractere *NULL* que marca o final da string. O programa a seguir, *strchr.c*, ilustra como usar a função *strchr*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
  char titulo[64] = "Bíblia do Programador C/C++, do Jamsa!";
  char *ptr;

  ptr = strchr(titulo, 'C');
    if (*ptr)
      printf("A primeira ocorrência de C está no deslocamento %d\n", ptr -
titulo);
   else
     printf("Caractere não encontrado\n");
 }
```

Quando você compilar e executar o programa *strchr.c*, sua tela exibirá o seguinte:

```
A primeira ocorrência de C está no deslocamento 22
C:\>
```

Você deve observar que *strchr* não contém um índice para a primeira ocorrência de um caractere; em vez disso, *strchr* contém um ponteiro para o caractere. Para lhe ajudar a compreender melhor a função *strchr*, considere a seguinte implementação:

```
char *strchr(const char *string, int letra)
 {
   while ((*string  != letra) && (*string))
      string++;
   return(string);
 }
```

177 RETORNANDO UM ÍNDICE À PRIMEIRA OCORRÊNCIA DE UMA STRING

Na Dica 176 você aprendeu como usar a função *strchr* para obter um ponteiro para a primeira ocorrência de um caractere dentro de uma string. No entanto, se você tratar as strings como matrizes, e não como ponteiros, provavelmente irá preferir trabalhar com um índice para o caractere, em vez de com um ponteiro. Você pode usar a função *strchr* para obter um índice para o caractere desejado subtraindo o endereço inicial da string do ponteiro que *strchr* retorna, como mostrado aqui:

```
char ptr = strchr(string, caractere);

indice = char_ptr - string;
```

Se *strchr* não encontrar o caractere na string, então o valor que strchr atribuir ao índice será igual ao tamanho da string. Além de usar strchr como detalhado nesta dica, você também pode usar a função *str_indice*, como mostrado aqui:

```
int str_indice(const char *string, int letra)
 {
   char *original = string;
   while ((*string != letra) && (*string))
       string++;
   return(string - original);
 }
```

ENCONTRANDO A ÚLTIMA OCORRÊNCIA DE UM CARACTERE EM UMA STRING 178

À medida que seus programas forem trabalhando com strings, algumas vezes você irá querer localizar a última ocorrência (a mais à direita) de um caractere específico dentro de uma string. Por exemplo, se você estiver trabalhando com uma string que contém um nome de caminho, poderá pesquisar na string o último caractere barra invertida (\) para localizar a posição onde começa o nome do arquivo. Para lhe ajudar a procurar a última ocorrência de um caractere dentro de uma string, a maioria dos compiladores fornece uma função chamada *strrchr*, que retornará um ponteiro para a última ocorrência de um caractere específico dentro de uma string, como mostrado aqui:

```
#include <string.h>

char *strrchr(const char *string, int caractere);
```

Se *strrchr* não encontrar o caractere que você especificar dentro da string, *strrchr* retornará um ponteiro para o caractere *NULL* que marcará o final da string. O programa a seguir, *strrchr.c*, ilustra como usar a função *strrchr*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char titulo[64] = "Bíblia do Programador C/C++, do Jamsa!";
   char *ptr;

   if (ptr = strrchr(titulo, 'C'))
     printf("A ocorrência mais à direita de C está no deslocamento %d\n",
     ptr - titulo);
   else
     printf("Caractere não encontrado\n");
 }
```

Observe que *strrchr* não contém um índice à última ocorrência de um caractere, mas, em vez disso, contém um ponteiro para o caractere. Para lhe ajudar a compreender melhor a função *strrchr*, considere a seguinte implementação:

```
char *strrchr (const char *string, int letra)
{
   char *ptr = NULL;

   while (*string)
     {
       if (*string == letra)
          ptr = string;
       string++;
     }
   return(ptr);
 }
```

179 RETORNANDO UM ÍNDICE À ÚLTIMA OCORRÊNCIA DE UMA STRING

Na Dica 178 você viu como usar a função *strrchr* para obter um ponteiro para a última ocorrência de um caractere dentro de uma string. No entanto, se você tratar uma string como uma matriz, e não como um ponteiro, poderá obter um índice para o caractere desejado subtraindo o endereço inicial da string do ponteiro que *strrchr* retornará:

```
char_ptr = strrchr(string, caractere);

indice = char_ptr - string;
```

Se *strrchr* não encontrar o caractere na string, o valor que *strrchr* atribuirá ao índice será igual ao tamanho da string. Além de usar *strrchr*, você poderá usar a função *strr_index*, como mostrado aqui:

```
int strr_index(const char *string, int letra)
 {
   char *original = string;
   char *ptr = NULL;

   while (*string)
     {
       if (*string == letra)
          ptr = string;
       string++;
     }
   return((*ptr) ? ptr-original: string-original);
 }
```

180 TRABALHANDO COM STRINGS DO TIPO FAR

Como será discutido na seção Gerenciamento da Memória no Windows mais à frente, os ponteiros *far* permitem que os programas do DOS acessem dados que residem fora do segmento atual de 64Kb. Quando você estiver trabalhando com ponteiros *far*, precisará também usar funções que esperam que seus parâmetros sejam ponteiros *far*. Infelizmente, nenhuma das rotinas de manipulação de strings que esta seção apresenta prevê ponteiros *far* para as strings. Passar um ponteiro *far* para uma das funções de manipulação de string que esta seção detalha provocará um erro. No entanto, para suportar os ponteiros *far*, muitos compiladores fornecem implementações de ponteiros *far* dessas funções. Por exemplo, para determinar o tamanho de uma string que um ponteiro *far* referencia, você poderia usar a função *_fstrlen*, como mostrado aqui:

```
#include <string.h>

size_t _fstrlen(const char *string)
```

Para determinar quais funções *far* seu compilador suporta, consulte a documentação do compilador

Nota: Como você aprendeu anteriormente, o Visual C++ não suporta as declarações far (sejam funções ou ponteiros), de modo que você pode usar a função strlen com os ponteiros char de qualquer tamanho no Visual C++.

181 ESCREVENDO FUNÇÕES STRING PARA AS STRINGS FAR

Na Dica 180 vimos que vários compiladores fornecem funções que suportam strings que os ponteiros *far* referenciam. Se seu compilador não fornece essas funções, você mesmo pode criar as funções de string *far* modificando as funçoes nesta seçao. Como um exemplo, a seguinte funçao, *fstreql*, ilustra a implementaçao baseada em ponteiros *far* de *streql* (em vez de a implementação padrão baseada em ponteiros locais):

```
int fstreql (char far *str1, char far *str2)
 {
   while ((*str1 == *str2) && (*str1))
     {
       str1++;
```

```
      str2++;
    }
  return((*str1 == NULL) && (*str2 == NULL));
}
```

Nota: Como você aprendeu anteriormente, o Visual C++ não suporta as declarações far, de modo que você poderá usar a função streql com os ponteiros char de qualquer tamanho no Visual C++.

CONTANDO O NÚMERO DE OCORRÊNCIAS DE UM CARACTERE EM UMA STRING 182

À medida que seus programas forem trabalhando com strings, algumas vezes você poderá querer saber o número de vezes que um caractere ocorre dentro de uma string. Para lhe ajudar a contar o número de vezes que um caractere ocorre dentro de uma string, seus programas podem usar a função *charcnt*, como mostrado aqui:

```
int charcnt(const char *string, int letra)
{
  int conta = 0;

  while (*string)
    if (*string == letra)
       conta++;
  return(conta);
}
```

INVERTENDO O CONTEÚDO DE UMA STRING 183

À medida que seus programas efetuarem diferentes operações, algumas vezes você poderá precisar inverter a ordem de caracteres dentro de uma string. Para simplificar tais operações, a maioria dos compiladores fornece uma função *strrev*, como mostrado aqui:

```
#include <string.h>

char *strrev(char *string);
```

Para compreender melhor a função *strrev*, considere a seguinte implementação:

```
char *strrev(char *string)
{
  char *original = string;
  char *frente = string;
  char temp;

  while (*string)
     string++;
  while (frente < string)
    {
      temp = *(--string);
      *string = *frente;
      *frente++ = temp;
    }
  return(original);
}
```

ATRIBUINDO UM CARACTERE ESPECÍFICO A UMA STRING INTEIRA 184

À medida que seus programas forem trabalhando com strings, é possível que às vezes você irá querer definir todos os caracteres em uma string com um caractere específico. Por exemplo, algumas vezes você quer sobrescrever o

valor atual de uma string antes de passar a string para uma função. Para simplificar a sobrescrita de todo caractere dentro de uma string, a maioria dos compiladores C fornece uma função *strset*, que atribui a cada caractere na string um caractere especificado, como mostrado aqui:

```
#include <string.h>

char *strset(char *string, int caractere);
```

A função *strset* atribui o caractere especificado para cada posição na string até encontrar o caractere *NULL*. Para compreender melhor a função *strset*, considere a seguinte implementação:

```
char *strset(char *string, int letra)
 {
   char *original = string;

   while (*string)
      *string++ = letra;
   return(original);
 }
```

Como você pode ver, a função percorre a string atribuindo o caractere especificado até que a função encontre o caractere *NULL*.

185 COMPARANDO DUAS STRINGS DE CARACTERES

Na Dica 173 você criou a função *streql*, que permite que seus programas testem se duas strings de caracteres são iguais. Dependendo do processamento que seu programa precisa executar, haverá algumas vezes (tais como quando seu programa efetuar uma operação de classificação) que você deverá saber se uma string é maior que outra. Para ajudar seus programas a efetuar operações que determinem o valor de várias strings, a maioria dos compiladores C fornece uma função chamada *strcmp*, que compara duas strings de caracteres, como mostrado aqui:

```
#include <string.h>

int strcmp(const char *str1, const *char str2);
```

Se as strings forem iguais, *strcmp* retornará o valor 0. Se a primeira string for maior que a segunda, *strcmp* retornará um valor menor que 0. Da mesma forma, se a segunda string for maior que a primeira, *strcmp* retornará um valor maior que 0. O programa a seguir, *strcmp.c*, ilustra como usar a função *strcmp*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   printf("Comparando Abc com Abc %d\n", strcmp("Abc", "Abc"));
   printf("Comparando abc com Abc %d\n", strcmp("abc", "Abc"));
   printf("Comparando abcd com abc %d\n", strcmp("abcd", "abc"));
   printf("Comparando Abc com Abcd %d\n", strcmp("Abc", "Abcd"));

   printf("Comparando abcd com abce %d\n", strcmp("abcd", "abce"));
   printf("Comparando Abce com Abcd %d\n", strcmp("Abce", "Abcd"));
 }
```

Para compreender melhor a função *strcmp*, considere a seguinte implementação:

```
int strcmp(const char *s1, const char *s2)
 {
   while ((*s1 == *s2) && (*s1))
     {
        s1++;
        s2++;
```

```
    }
  if ((*s1 == *s2) && (! *s1))     // Strings iguais
     return(0);
  else if ((*s1) && (! *s2))     // Iguais mas s1 maior
     return(-1);
  else if ((*s2) && (! *s1))       // Iguais mas s2 maior
     return(1);
  else
     return((*s1 > *s2) ? -1: 1);  // Diferentes
}
```

COMPARANDO OS PRIMEIROS N CARACTERES DE DUAS STRINGS 186

Na D.ca 185 você aprendeu como usar a função *strcmp* para comparar duas strings. Dependendo da função do seu programa, algumas vezes você poderá querer somente comparar os primeiros *n* caracteres de duas strings. Para facilitar a comparação somente de *n* caracteres de duas strings, a maioria dos compiladores C fornece uma função chamada *strncmp*, como mostrado aqui:

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

Como *strcmp*, a função *strncmp* retornará o valor 0 se as strings forem iguais, e um valor menor ou maior que 0, dependendo se a primeira ou a segunda string for maior. O programa a seguir, *strncmp.c*, ilustra como usar a função *strncmp*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   printf("Comparando 3 letras Abc com Abc %d\n", strncmp("Abc", "Abc", 3));
   printf("Comparando 3 letras abc com Abc %d\n", strncmp("abc", "Abc", 3));
   printf("Comparando 3 letras abcd com abc %d\n", strncmp("abcd", "abc", 3));
   printf("Comparando 5 letras Abc com Abcd %d\n", strncmp("Abc", "Abcd", 5));
   printf("Comparando 4 letras abcd com abce %d\n", strncmp("abcd", "abce", 4));
 }
```

Para compreender melhor a função *strncmp*, considere a seguinte implementação:

```
int strncmp(const char *s1, const char *s2, int n)
 {
   int i = 0;

   while ((*s1 == *s2) && (*s1) && i < n)
     {
        s1++;
        s2++;
        i++;
     }
   if (i == 0)                    // Strings iguais
      return(0);
   else if ((*s1 == *s2) && (! *s1))  // Strings iguais
      return(0);
   else if ((*s1) && (! *s2))      // Iguais, mas s1 maior
      return(-1);
   else if ((*s2) && (! *s1))       // Iguais, mas s2 maior
      return(1);
   else
      return((*s1 > *s2) ? -1: 1);
```

```
}
```

187 COMPARANDO STRINGS SEM CONSIDERAR A CAIXA

Na Dica 185 você viu como usar a função *strcmp* para comparar duas strings. Da mesma forma, na Dica 186 você viu como usar a função *strncmp* para comparar os primeiros *n* caracteres de duas strings. Tanto *strcmp* e *strncmp* consideram as letras maiúsculas e minúsculas como distintas. Dependendo da função que seu programa executa, algumas vezes você pode querer que a comparação de strings ignore a caixa. Para tais operações, a maioria dos compiladores C fornece as funçoes *stricmp* e *strncmpi*, como mostrado aqui:

```
#include <string.h>

int stricmp(const char s1, const char s2);
int strncmpi(const char *s1, const char *s2, size_t n);
```

O programa a seguir, *cmpcaixa.c*, ilustra como usar as funções *stricmp* e *strncmpi*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   printf("Comparando Abc com Abc %d\n", stricmp("Abc", "Abc"));
   printf("Comparando abc com Abc %d\n", stricmp("abc", "Abc"));
   printf("Comparando 3 letras abcd com ABC %d\n", strncmpi("abcd", "ABC", 3));
   printf("Comparando 5 letras abc com Abcd %d\n", strncmpi("abc", "Abcd", 5));
 }
```

Quando você compilar e executar o programa *cmpcaixa.c*, sua tela exibirá a seguinte saída:

```
Comparando ABC com ABC 0
Comparando abc com Abc 0
Comparando 3 letras abcd com ABC 0
Comparando 5 letras abc com Abcd -1
C:\>
```

188 CONVERTENDO A REPRESENTAÇÃO EM STRING DE CARACTERES DE UM NÚMERO

Quando seus programas trabalham com strings, uma das operações mais comuns que você precisa executar é converter uma representação ASCII de um valor para um valor numérico. Por exemplo, se você pedir que o usuário digite seu salário, poderá precisar converter a entrada em string de caracteres em um valor em ponto flutuante. Para lhe ajudar a converter valores ASCII, a maioria dos compiladores C oferece um conjunto de funções de biblioteca de execução que efetuam a conversao de ASCII para numérico. A Tabela 188 descreve resumidamente as funções padrão de conversão ASCII.

Tabela 188 Funções da biblioteca de execução que seus programas podem usar para converter representações ASCII de um valor numérico.

Função	Propósito
atof	Converte a representação em string de caracteres de um valor em ponto flutuante
atoi	Converte a representação em string de caracteres de um valor inteiro
atol	Converte a representação em string de caracteres de um valor inteiro longo
strtod	Converte a representação em string de caracteres de um valor em precisão dupla
strtol	Converte a representação em string de caracteres de um valor longo

O programa a seguir *asciinum.c*, ilustra como usar as funções de ASCII para numérico:

```
#include <stdio.h>
#include <stdlib.h>
```

```
void main(void)
 {
   int int_result;
   float float_result;
   long long_result;

   int_result = atoi("1234");
   float_result = atof("12345.678");
   long_result = atol("1234567L");
   printf("%d %f %ld\n", int_result, float_result, long_result);
 }
```

DUPLICANDO O CONTEÚDO DE UMA STRING 189

Quando seus programas trabalham com strings, algumas vezes você pode querer duplicar o conteúdo de uma string rapidamente. Se às vezes seu programa precisar copiar a string e outras vezes não, você poderá querer que o programa aloque memória *dinamicamente* (durante a execução do programa) para conter a cópia da string conforme necessário. Para permitir que seus programas aloquem memória durante a execução (dinamicamente) para criar uma cópia da string de caracteres, a maioria dos compiladores C fornece a função *strdup*, como mostrado aqui:

```
#include <string.h>

char *strdup(const char *alguma_string);
```

Quando você chama *strdup*, a função usa *malloc* para alocar memória, e, depois, copia o conteúdo da string na posição de memória. Quando seu programa terminar de usar a cópia da string, ele poderá liberar a memória usando o comando *free*. O programa a seguir, *strdup.c*, ilustra como usar a função *strdup*:

```
#include <stdio.h>
#include <string.h>

void main(void)
{
   char *titulo;

   if ((titulo = strdup("Bíblia do Programador C/C++, do Jamsa!")))
     printf("Titulo: %s\n", titulo);
   else
     printf("Erro ao duplicar a string");
}
```

Para compreender melhor a função *strdup*, considere a seguinte implementação:

```
#include <string.h>
#include <malloc.h>

char *strdup(const char *s1)
 {
   char *ptr;

   if ((ptr = malloc(strlen(s1)+1)))   // Aloca buffer
      strcpy(ptr, s1);
   return(ptr);
 }
```

ENCONTRANDO UM CARACTERE DA PRIMEIRA OCORRÊNCIA DE UM DETERMINADO CONJUNTO 190

Na Dica 176 você aprendeu como usar a função *strchr* para localizar a primeira ocorrência de um caractere específico. Dependendo da função que seu programa executar, algumas vezes você irá querer pesquisar em uma

string a primeira ocorrência de qualquer caractere de um dado conjunto de caracteres. Para lhe ajudar a procurar em uma string qualquer caractere de um conjunto, a maioria dos compiladores C fornece a função *strspn*, como mostrado aqui:

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

Dentro da string *s1*, a função retorna o deslocamento do primeiro caractere não contido dentro da string *s2*. O programa a seguir, *strspn.c*, ilustra como usar a função *strspn*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   printf("Procurando Abc em AbcDef %d\n", strspn("AbcDef", "Abc"));
   printf("Procurando cbA em AbcDef %d\n", strspn("AbcDef", "cbA"));
   printf("Procurando Def em AbcAbc %d\n", strspn("AbcAbc", "Def"));
 }
```

Quando você compilar e executar o programa *strspn.c*, sua tela exibirá o seguinte:

```
Procurando Abc em AbcDef 3
Procurando cbA em AbcDef 3
Procurando Def em AbcAbc 0
C:\>
```

Para compreender melhor *strspn*, considere a seguinte implementação:

```
size_t strspn(const char *s1, const char *s2)
 {
   int i, j;

   for (i = 0; *s1; i++, s1++)
     {
       for (j = 0; s2[j]; j++)
         if (*s1 == s2[j])
             break;
       if (s2[j] == NULL)
             break;
     }
   return(i);
 }
```

191 LOCALIZANDO UMA SUBSTRING DENTRO DE UMA STRING

À medida que seus programas trabalharem com strings, algumas vezes você precisará procurar uma substring específica dentro de uma string. Para lhe ajudar a procurar uma substring em uma string, a maioria dos compiladores C fornece uma função chamada *strstr*, como mostrado aqui:

```
#include <string.h>

strstr(string, substring);
```

Se a substring existir dentro da string, *strstr* retornará um ponteiro para a primeira ocorrência da string. Se *strstr* não encontrar a substring, a função retornará *NULL*. O programa a seguir, *strstr.c*, ilustra como usar *strstr*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
```

```
    printf("Procurando Abc em AbcDef %s\n",
       (strstr("AbcDef", "Abc")) "Encontrado" : "Não encontrado");
    printf("Procurando Abc em abcDef %s\n",
       (strstr("abcDef", "Abc")) ? "Encontrado": "Não encontrado");
    printf("Procurando Abc em AbcAbc %s\n",
       (strstr("AbcAbc", "Abc")) ? "Encontrado": "Não encontrado");
}
```

Para lhe ajudar a compreender melhor *strstr*, considere a seguinte implementação:

```
char *strstr(const char *s1, const char *s2)
{
  \int i, j, k;
  for (i = 0; s1[i]; i++)
     for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
         if (! s2[k+1])
            return(s1 + i);
  return(NULL);
}
```

CONTANDO O NÚMERO DE OCORRÊNCIAS DA SUBSTRING 192

Na Dica 191 você viu como usar a função *strstr* para localizar uma substring dentro de uma string. Em certas ocasiões, você poderá querer saber o número de vezes que uma substring aparece dentro de uma string. A seguinte função *strstr_cnt*, permite-lhe determinar quantas vezes uma determinada substring ocorre dentro de uma string:

```
int strstr cnt(const char *string, const char *substring)
{
  int i, j, k, conta = 0;
  for (i = 0; string[i]; i++)
     for (j = i, k - =0; string[j]== substring[k]; j++, k++)
        if (! substring[k + 1])
           conta++;
  return(conta);
}
```

OBTENDO UM ÍNDICE PARA UMA SUBSTRING 193

Na Dica 191 você aprendeu como usar a função *strstr* para obter um ponteiro para uma substring dentro de uma string. Se você tratar as strings de caracteres como matrizes, em vez de usar ponteiros, algumas vezes irá querer saber o índice do caractere onde uma substring inicia dentro da string. Usando o valor que *strstr* retorna, você pode subtrair o endereço da string para produzir um índice:

```
indice = strstr(string, substr) - string;
```

Se *strstr* não encontrar a substring, o valor do índice será igual ao tamanho da string. Além disso, seus programas podem usar a função *substring_index* para obter um índice para uma substring, como mostrado aqui:

```
int substring_index(const char *s1, const char *s2)
{
  int i, j, k;
  for (i = 0; s1[i]; i++)
     for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
        if (! s2[k+1])
           return(i);
  return(i);
}
```

OBTENDO A OCORRÊNCIA MAIS À DIREITA DE UMA SUBSTRING 194

Na Dica 191 você usou a função *strstr* para determinar a primeira ocorrência de uma substring dentro de uma string. Dependendo da função do seu programa, algumas vezes você poderá querer conhecer a última (a mais a

direita) ocorrência de uma substring dentro de uma string. A função a seguir, *r_strstr*, retorna um ponteiro para a ocorrência mais à direita de uma substring dentro de uma string, ou o valor *NULL* se a substring não existir:

```
char *r_strstr(const char *s1, const char *s2)
 {
   int i, j, k, esquerda = 0;

   for (i = 0; s1[i]; i++)
      for (j = i, k = 0; s1[j] == s2[k]; j++, k++)
         if (! s2[k+1])
            esquerda = i;
   return((esquerda) ? s1+esquerda: NULL);
 }
```

195 EXIBINDO UMA STRING SEM O ESPECIFICADOR DE FORMATO %S

Várias dicas nesta seção usaram o especificador de formato para exibir strings de caracteres. O comando a seguir, por exemplo, usa *printf* para exibir o conteúdo da variável string de caractere chamada *titulo*:

```
printf("%s", titulo);
```

O primeiro argumento passado ao comando *printf* é uma string de caracteres, que pode conter um ou mais especificadores de formato. Quando seus programas usam *printf* para exibir somente uma string de caracteres, como mostra o exemplo anterior, você pode omitir a string de caracteres que contém o especificador de formato e passar a *printf* a string de caracteres que você quer exibir, como mostrado aqui:

```
printf(titulo);
```

Como você pode ver, o primeiro argumento de *printf* não é nada mais que uma string de caracteres que contém um ou mais símbolos especiais.

196 REMOVENDO UMA SUBSTRING DE DENTRO DE UMA STRING

Na Dica 191 você usou a função *strstr* para determinar a posição inicial de uma substring dentro de uma string. Em muitos casos, seu programa precisa remover uma substring de dentro de uma string. Para fazer isso, você pode usar a função *strstr_rem*, que remove a primeira ocorrência de uma substring, como mostrado aqui:

```
char *strstr_rem(char *string, char *substring)
 {
   int i, j, k, pos = -1;

   for (i = 0; string[i] && (pos == -1); i++)
      for (j = i; k = 0; str[j] == substring[k]; j++, k++)
         if (! substring[k + 1])
            pos = i;
   if (pos != -1)    // A substring foi encontrada
     {
        for (k = 0; substr[k]; k++)
           ;
        for (j = loc, i = loc + k, string[i]; j++, i++)
           string[j] = string[i];
        string[i] == NULL;
     }
   return(string);
 }
```

SUBSTITUINDO UMA SUBSTRING POR OUTRA 197

Na Dica 196 você usou a função *strstr_rem* para remover uma substring de dentro de uma string. Em muitos casos, seus programas precisam substituir a primeira ocorrência de uma substring por outra substring. Você pode fazer isso com a seguinte função, *strstr_rep*, como mostrado aqui:

```
#include <string.h>

char *strstr_rep(char *origem, char *antigo, char *novo)
 {
   char *original = origem;
   char temp[256];
   int  tam_antigo = strlen(antigo);
   int i, j, k, posicao = -1;

   for (i = 0; origem[i] && (posicao == -1); ++i)
     for (j = i; k = 0; origem[j] == antigo[k]; j++, k++)
        if (! antigo[k+1])
           posicao = i;
   if (posicao != -1)
     {
       for (j = 0; j < posicao; j++)
          temp[j] = origem[j];
       for (i = 0; novo[i]; i++, j++)
          temp[j] = novo[i];
       for (k = posicao + tam_antigo; origem[k]; k++, j++)
          temp[j] = origem[k];
       temp[j] = NULL;
       for (i = 0; origem[i] = temp[i]; i++);  // Laço NULL
     }
   return(original);
 }
```

CONVERTENDO UMA REPRESENTAÇÃO NUMÉRICA ASCII 198

Quando seus programas trabalham com strings de caracteres, freqüentemente precisam converter uma representação ASCII de um valor, tal como 1.2345, ao valor *int, float, double, long* ou *unsigned* correspondente. Para lhe ajudar a efetuar essas operações, C fornece as funções definidas na Tabela 198.

Tabela 198 As funções que você pode usar para converter representações em ASCII para numérico.

Função	Propósito
atof	Converte uma representação ASCII de um valor do tipo *float*
atoi	Converte uma representação ASCII de um valor do tipo *int*
atol	Converte uma representação ASCII de um valor do tipo *long int*

Os formatos das funções detalhadas na Tabela 198 são como segue:

```
#include <stdlib.h>

double atof(char *string);
int atoi(char *string);
int atol(char *string);
```

Se uma função não puder converter a string de caracteres para um valor numérico, a função retornará 0. O programa a seguir, *asc_para.c*, ilustra como usar as funções *ato:*

```
#include <stdio.h>
#include <stdlib.h>

void main (void)
```

```
{
  int int_valor;
  float flt_valor;
  long long_valor;

  int_valor = atoi("12345");
  flt_valor = atof("33.45");
  long_valor = atol("12BAD");
  printf("int %d float %5.2f long %ld\n", int_valor, flt_valor, long valor);
}
```

Quando você compilar e executar o programa *asc_para.c*, sua tela exibirá a seguinte saída:

```
int 12345 float 33.45 long 12
C:\>
```

Observe a chamada que o programa faz a *atol*. Como você pode ver, quando a função encontra o valor não-numérico (a letra B), a função finaliza a conversão, retornando o valor que a função já converteu até aquele ponto.

199 DETERMINANDO SE UM CARACTERE É ALFANUMÉRICO

Um caractere alfanumérico é uma letra ou um dígito. Em outras palavras, um caractere alfanumérico é uma letra maiúscula de A até Z, uma letra minúscula de a até z, ou um dígito de 0 até 9. Para ajudar seus programas a determinar se um caractere é alfanumérico, o arquivo de cabeçalho *ctype.h* contém uma macro chamada *isalnum*. A macro examinará uma letra e retornará o valor 0 se o caractere não for alfanumérico, e um valor diferente de zero para caracteres alfanuméricos, como mostrado aqui:

```
if (isalnum(letra))
```

Para compreender melhor a macro *isalnum*, considere a seguinte implementação:

```
#define isalnum(c) ((toupper((c)) >= 'A' && (toupper((c)) <= 'Z' || ((c)
>= '0' && ((c) <= '9'))
```

200 DETERMINANDO SE UM CARACTERE É UMA LETRA

À medida que seus programas trabalharem com caracteres dentro de strings, algumas vezes precisarão testar se um caractere contém uma letra do alfabeto (maiúscula ou minúscula). Para ajudar seus programas a determinar se um caractere é uma letra do alfabeto, o arquivo de cabeçalho *ctype.h* fornece a macro *isalpha*. A macro examina uma letra e retorna o valor 0 se o caractere não contém uma letra maiúscula de A até Z ou minúscula de a até z. Se o caractere contém uma letra do alfabeto, então a macro retorna um valor diferente de zero:

```
if (isalpha (caractere))
```

Para compreender melhor a macro *isalpha*, considere a seguinte implementação:

```
#define isalpha(c) (toupper((c)) >= 'A' && (toupper((c)) <= 'Z')
```

201 DETERMINANDO SE UM CARACTERE CONTÉM UM VALOR ASCII

Um valor ASCII é um valor no intervalo de 0 até 127. Quando seus programas trabalham com os caracteres de uma string, algumas vezes você precisa determinar se um caractere contém um valor ASCII. Para ajudar seus programas a determinar um valor ASCII, o arquivo de cabeçalho *ctype.h* contém a macro *isascii*, que examina uma letra e retorna o valor 0 caso o caractere não contenha um ASCII, e um valor diferente de 0 caso o caractere contenha um valor ASCII, como mostrado aqui:

```
if (isascii (caractere))
```

Para compreender melhor a macro *isascii*, considere a seguinte implementação:

```
#define isascii (ltr)  ((unsigned) (ltr) < 128)
```

Como você pode ver, a macro *isascii* considera um valor no intervalo de 0 até 127 como ASCII.

DETERMINANDO SE UM CARACTERE É UM CARACTERE DE CONTROLE 202

Um *caractere de controle* é um valor de ^A até ^Z ou de ^a até ^z. Aplicativos diferentes usam os caracteres de controle de forma diferente. Por exemplo, o DOS usa o caractere Ctrl+Z para representar o fim de um arquivo. Outros processadores de texto usam caracteres de controle para representar negrito ou itálico. Quando você trabalha com caracteres em uma string, algumas vezes precisa determinar se um caractere é um caractere de controle. Para ajudar seus programas a efetuar esse teste, o arquivo de cabeçalho *ctype.h* contém a macro *iscntrl*, que retorna um valor diferente de zero para um caractere de controle e 0 se a letra não for um caractere de controle, como mostrado aqui:

```
if (iscntrl(caractere))
```

DETERMINANDO SE UM CARACTERE É UM DÍGITO 203

Um *dígito* é um valor ASCII de 0 até 9. Quando você trabalha com strings, algumas vezes precisa determinar se um caractere é um dígito. Para lhe ajudar a testar se um caractere é um dígito, o arquivo de cabeçalho *ctype.h* fornece a macro *isdigit*. A macro *isdigit* examina um caractere e retorna o valor 0 se o caractere não for um dígito, e um valor diferente de 0 para caracteres no intervalo de 0 a 9, como mostrado aqui:

```
if (isdigit (letra))
```

Para compreender melhor a macro *isdigit*, considere a seguinte implementação:

```
#define isdigit(c) ((c) >= '0' && (c) <= '9')
```

DETERMINANDO SE UM CARACTERE É UM CARACTERE GRÁFICO 204

Um *caractere gráfico* é um caractere imprimível (veja *isprint*), excluindo o caractere de espaço (ASCII 32). Quando seus programas executam operações de saída de caractere, algumas vezes você pode querer saber se um caractere é um caractere gráfico. Para ajudar seus programas a efetuar esse teste, o arquivo de cabeçalho *ctype.h* fornece a macro *isgraph*. A macro *isgraph* examina um caractere e retorna o valor 0 se o caractere não for um gráfico, e um valor diferente de 0 para caracteres gráficos:

```
if (isgraph (letra))
```

Para compreender melhor a macro *isgraph*, considere a seguinte implementação:

```
#define isgraph (ltr) ((ltr) >= 32) && ((ltr) <= 127)
```

Como você pode ver, um caractere gráfico é qualquer caractere ASCII no intervalo de 33 a 127.

DETERMINANDO SE UM CARACTERE É MAIÚSCULO OU MINÚSCULO 205

À medida que seus programas forem trabalhando com caracteres dentro de uma string, algumas vezes você precisará saber se um caractere é uma letra maiúscula ou minúscula. Para ajudar seus programas a testar a caixa das letras, o arquivo de cabeçalho *ctype.h* fornece as macros *islower* e *isupper*. Essas macros examinam um caractere e retornam um valor 0 para caracteres que não são minúsculos (*islower*) ou maiúsculos (*isupper*) e um valor diferente de 0 em caso contrário.

```
if (islower (caractere))

if (isupper (caractere))
```

Para compreender melhor as macros *islower* e *isupper*, considere as seguintes implementações:

```
#define islower(c) ((c) >= 'a' && (c) <= 'z')
#define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

206 DETERMINANDO SE UM CARACTERE É IMPRIMÍVEL

Quando seus programas efetuam saída de caracteres, você pode querer examinar cada caractere para garantir que somente *caracteres imprimíveis* sejam exibidos. Um caractere imprimível é qualquer caractere no intervalo de 32 (o espaço) até 127 (o caractere Del). Para ajudar seus programas a testar se um caractere é imprimível, o arquivo de cabeçalho *ctype.h* fornece a macro *isprint*. A macro *isprint* retorna um valor diferente de 0 para caracteres imprimíveis, e um valor 0 para caracteres que não são imprimíveis:

```
if (isprint (caractere))
```

Para compreender melhor a macro *isprint*, considere a seguinte implementação:

```
#define isprint (ltr) ((ltr) >= 32) ((ltr) <= 127)
```

Como você pode ver, a macro *isprint* considera qualquer caractere ASCII no intervalo de 32 até 127 como um caractere imprimível.

207 DETERMINANDO SE UM CARACTERE É UM SINAL DE PONTUAÇÃO

Em um texto dentro de um livro, os sinais de pontuação incluem vírgulas, ponto-e-vírgula, ponto, sinais de interrogação etc. No entanto, dentro de C, um sinal de pontuação é qualquer caractere gráfico ASCII que não seja alfanumérico. À medida que seus programas trabalharem com caracteres em uma string, algumas vezes você precisará testar se um caractere contém um sinal de pontuação. Para ajudar seus programas a testar sinais de pontuação, o arquivo de cabeçalho *ctype.h* define a macro *ispunct*. Essa macro examina um caractere e retorna um valor diferente de 0 para um caractere que contém um sinal de pontuação e um valor 0 para um caractere que não contém um sinal de pontuação:

```
if (ispunct (caractere))
```

Para compreender melhor a macro *ispunct*, considere a seguinte implementação.

```
#define ispunct(c) (isgraph(c)) && ! isalphanum((c)))
```

208 DETERMINANDO SE UM CARACTERE CONTÉM ESPAÇO EM BRANCO

O termo *caracteres de espaço em branco* inclui os seguintes caracteres: espaço, tabulação, retorno do carro, nova linha, tabulação vertical e alimentação de formulário. Quando seus programas efetuam saída de caracteres, algumas vezes você precisa testar se um caractere contém um espaço em branco. Para ajudar seus programas a testar o espaço em branco, o arquivo de cabeçalho *ctype.h* fornece a macro *isspace*. Essa macro examina um caractere e retorna um valor diferente de 0 para caracteres de espaço em branco, e um valor 0 para caracteres de não espaço em branco:

```
if (isspace (caractere))
```

Para compreender melhor a macro *isspace*, considere a seguinte implementação:

```
#define isspace(c) (((c) == 32) || ((c) == 9) || ((c) == 13))
```

DETERMINANDO SE UM CARACTERE É UM VALOR HEXADECIMAL 209

Um *valor hexadecimal* é um dígito no intervalo de 0 até 9 ou uma letra da maiúscula A até F ou da minúscula a até f. Quando seus programas trabalham com caracteres em uma string, algumas vezes você precisa determinar se um caractere contém um dígito hexadecimal. Para ajudar seus programas a testar se um dígito é hexadecimal, o arquivo de cabeçalho *ctype.h* define a macro *isxdigit*. Essa macro examina um caractere e retorna um valor diferente de 0 se o caractere for um dígito hexadecimal, e um valor 0 se o caractere não for hexadecimal.

```
if (isxdigit (caractere))
```

Para compreender melhor a macro *ixdigit*, considere a seguinte implementação:

```
#define isxdigit(c) (isnum((c)) || (toupper((c)) >= 'A' && toupper((c)) <=
'F'))
```

CONVERTENDO UM CARACTERE PARA MAIÚSCULA 210

À medida que você trabalha com strings de caracteres, uma operação comum que seus programas precisam executar é converter letras minúsculas para maiúsculas. Quando você quiser converter a caixa das letras, seus programas têm duas escolhas. Eles podem usar a macro *_toupper*, que está definida no arquivo de cabeçalho *ctype.h*, ou podem usar a função da biblioteca de execução *toupper*. Os formatos da macro e da função são como seguem:

```
#include <ctype.h>

int _toupper (int caractere);
int toupper (int caractere);
```

Embora tanto a macro quanto a função convertam um caractere para maiúscula, elas trabalham de forma diferente. A macro *_toupper* não testa se o caractere que está convertendo é minúsculo. Se você chamar a macro com um caractere que não é minúsculo, ela causará um erro. Por outro lado, a função *toupper* somente converte letras minúsculas, e deixa todos os outros caracteres inalterados. Se você tiver certeza de que o caractere contém uma letra minúscula, use a macro *_toupper*; essa macro executa mais rapidamente que a função. No entanto, se você não tiver certeza se o caractere é minúsculo, use a função *toupper*. O programa a seguir, *toupper.c*, ilustra o uso de *_toupper* e *toupper*, bem como os erros que podem ocorrer quando você usa a macro com caracteres que não são minúsculos:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
 {
   char string[]= "Biblia do Programador C/C++, do Jamsa!";
   int i;

   for (i = 0; string[i]; i++)
      putchar(toupper(string[i]));
   putchar('\n');
   for (i = 0; string[i]; i++)
      putchar(_toupper(string[i]));
   putchar('\n');
 }
```

Quando você compilar e executar este programa, sua tela exibirá a primeira string (usando *toupper*) nas letras maiúsculas corretas. No entanto, a segunda string conterá caracteres não padrão (símbolos, gráficos etc.), pois *_toupper* tenta converter caracteres maiúsculos como faz com os caracteres minúsculos.

211 CONVERTENDO UM CARACTERE PARA MINÚSCULO

À medida que você trabalha com strings de caracteres, uma operação comum que seus programas precisam executar é converter um caractere de maiúsculo para minúsculo. Quando você quiser converter a caixa, seus programas têm duas escolhas. Podem usar a macro *_tolower*, que está definida no arquivo de cabeçalho *ctype.h*, ou podem usar a função da biblioteca de execução *tolower*. Os formatos da macro e da função são estes:

```
#include <ctype.h>

int _tolower (int caractere);
int tolower (int caractere);
```

Embora tanto a macro quanto a função convertam um caractere para minúsculo, elas trabalham de forma diferente. A macro *_tolower* não testa se o caractere que está convertendo é maiúsculo. Se você chamar a macro com um caractere que não é maiúsculo, ela causará um erro. Por outro lado, a função *tolower* somente converte letras maiúsculas, e deixa todos os outros caracteres inalterados. Se você tiver certeza de que o caractere contém uma letra maiúscula, use a macro *_tolower*, essa macro é mais rápida que a função. No entanto, se você não tiver certeza se o caractere é maiúsculo, use a função *tolower*. O programa a seguir, *tolower.c*, ilustra o uso de *_tolower* e *tolower*, bem como os erros que podem ocorrer quando você usa a macro com caracteres que não são maiúsculos:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
 {
   char string[] = "Bíblia do Programador C/C++, do Jamsa!";
   int i;

   for (i = 0; string[i]; i++)
     putchar(tolower(string[i]));
   putchar('\n');
   for (i = 0; string[i]; i++)
     putchar(_tolower(string[i]));
   putchar('\n');
 }
```

Quando você compilar e executar este programa, sua tela exibirá a primeira string (usando *tolower*) em letras minúsculas corretas. No entanto, a segunda string conterá caracteres não-padrão (símbolos, gráficos e assim por diante), pois *_toupper* tenta converter caracteres maiúsculos como faz com os caracteres minúsculos.

212 TRABALHANDO COM CARACTERES ASCII

Quando você trabalha com strings de caracteres e diferentes funções de caracteres, algumas vezes precisa garantir que um caractere seja um caractere ASCII válido; isto é, que o valor está no intervalo de 0 até 127. Para garantir que um caractere é um caractere ASCII válido, você pode usar a macro *toascii*, que está definida no arquivo de cabeçalho *ctype.h*, como mostrado aqui:

```
#include <ctype.h>

int toascii (int caractere)
```

Para compreender melhor a macro *toascii*, considere a seguinte implementação:

```
#define toascii (caractere) ((caractere) & 0x7F)
```

Para melhorar o desempenho, a macro *toascii* efetua uma operação *E bit a bit* que zera o bit mais significativo do byte de valor do caractere. A operação *E bit a bit* ajuda a macro a garantir que o valor está no intervalo de 0-127.

ESCREVENDO SAÍDA FORMATADA EM UMA VARIÁVEL STRING 213

Como você sabe, a função *printf* lhe permite escrever saída formatada na tela. Dependendo dos requisitos do seu programa, algumas vezes você precisa trabalhar com uma string de caracteres que contém saída formatada. Por exemplo, digamos que seus funcionários tenham um número funcional de 5 dígitos e um identificador de 3 letras para a região (tal como Rec, para Recife). Assuma o fato de você armazenar informações sobre cada funcionário em um arquivo nomeado com uma combinação desses dois valores (tal como REC12345). A função *sprintf* lhe permite escrever saída formatada em uma string de caracteres. O formato da função *sprintf* é:

```
#include <stdio.h>

int sprintf (char *string, const char *formato [,argumentos...]);
```

O programa a seguir, *sprintf.c*, usa a função *sprintf* para criar um nome de arquivo de funcionários de 8 caracteres:

```
#include <stdio.h>

void main(void)
 {
   int func_numero = 12345;
   char regiao[] = "REC";
   char nomearq[64];

   sprintf(nomearq, "%s%d", regiao, func_numero);
   printf("Nome do arquivo do func: %s\n", nomearq);
 }
```

LENDO A ENTRADA DE UMA STRING DE CARACTERES 214

Como você aprendeu, a função *scanf* lhe permite ler a entrada formatada de *stdin*. Dependendo do processamento do seu programa, algumas vezes uma string de caracteres conterá campos que você quer atribuir a variáveis específicas. A função *sscanf* permite que seus programas leiam valores de uma string e atribuam os valores a variáveis específicas. O formato da função *sscanf* é como segue:

```
#include <stdio.h>

int sscanf (const char *string, const char *formato [,argumentos]);
```

Os argumentos que seu programa passa para a função *sscanf* precisam ser ponteiros para endereços de variáveis. Se *sscanf* atribuir campos com sucesso, ela retornará o número de campos que atribuiu. Se *sscanf* não atribuir campos, então retornará *0* ou *EOF* caso tenha encontrado o final da string. O programa a seguir, *sscanf.c*, ilustra a função *sscanf*.

```
#include <stdio.h>

void main(void)
 {
   int idade;
   float salario;
   char string[] = "33 2500.00";

   sscanf(string, "%d %f\n", &idade, &salario);
   printf("Idade: %d Salário %f\n", idade, salario);
 }
```

215 "TOKENIZANDO" AS STRINGS PARA POUPAR ESPAÇO

Tokenizar strings é o processo de usar um valor único para representar uma string. Por exemplo, assuma que você tenha um programa que trabalhe com um grande número de strings de caracteres. Digamos que o programa contenha um banco de dados das contas dos seus clientes por cidade e por estado. Dependendo de como o programa efetua seu processamento, você poderia terminar com muitos testes diferentes, como ilustra o programa a seguir:

```
if (streql (cidade, "Recife"))
    // Comando
else if (streql (cidade, "Salvador"))
    // Comando
else if (streql (cidade, "Rio"))
    // Comando
```

Dentro de cada uma das funções do seu programa que executa teste repetitivo *else if*, o programa consome uma considerável quantidade de espaço para as constantes strings, bem como uma quantidade considerável de tempo realizando comparações de strings. Em vez de usar chamadas de strings repetitivas, você pode criar uma função chamada *tokeniza_string*, que retorna um token (um símbolo) distinto para cada string. Dentro da função de exemplo, o teste de seu programa se tornará o seguinte:

```
int cidade_token;

cidade_token = tokeniza_string(cidade);
if (cidade_token == Recife_token)
    // Comando
else if (cidade_token == Salvador_token)
    // Comando
else if (cidade_token == Rio_token)
    // Comando
```

Usar tokens desse modo ajuda a eliminar a quantidade de espaço de dados que as consoantes da string consomem. Eliminar as comparações de string também aumenta o desempenho do programa.

216 INICIALIZANDO UMA STRING

Na seção Matrizes, Ponteiros e Estruturas mais à frente, você aprenderá como atribuir valores às matrizes quando seu programa declarar a matriz. C representa as strings de caracteres como uma matriz de bytes. Quando você declara uma string, geralmente especifica um valor inicial, como mostrado aqui:

```
char titulo[] = "Bíblia do Programador C/C++, do Jamsa!";
char secao[64] = "Strings";
```

Na string *titulo*, o compilador C alocará uma matriz grande o suficiente para conter os caracteres especificados (mais o caractere *NULL*). Como a string "Bíblia do Programador C/C++, do Jamsa!" tem 38 caracteres de tamanho, a string *titulo* pode conter 38 caracteres imprimíveis mais o caractere *NULL*. Se você, mais tarde, atribuir mais de 39 caracteres à string, sobrescreverá a memória que armazena o valor de outra variável. Na string *secao*, o compilador alocará uma string que pode armazenar 64 caracteres. O compilador atribuirá aos primeiros sete bytes da string as letras na palavra "Strings", e ao oitavo byte o caractere *NULL*. O compilador normalmente inicializa os 56 caracteres restantes como *NULL*.

217 COMPREENDENDO AS FUNÇÕES

A maioria dos programas apresentados até agora neste livro usa somente a função *main*. À medida que seus programas vão se tornando maiores e mais complexos, você pode simplificar seu trabalho e melhorar a clareza do programa dividindo o programa em partes menores, chamadas *funções*. Por exemplo, assuma que você esteja criando um programa de contabilidade. Você poderia ter uma função que efetuasse as operações normais de um contador, uma função diferente para as contas a pagar, uma terceira para as contas a receber e uma quarta para gerar um balanço. Se você colocasse todos os comandos do programa dentro de *main*, seu programa ficaria muito grande, e seria difícil de entender. À medida que o tamanho e a complexidade do programa aumentam, aumenta

também a possibilidade de erros. Se você dividir o programa em blocos menores e mais facilmente gerenciáveis, poderá evitar os erros. Uma função é uma coleção nomeada de comandos que efetuam uma tarefa específica. Por exemplo, a função a seguir, *oi_pessoal*, usa *printf* para exibir uma mensagem:

```
void oi_pessoal (void)
 {
   printf("Oi pessoal!\n");
 }
```

A palavra-chave *void* diz a C que a função não retorna um valor. Em muitos casos, suas funções usarão *return* para retornar o resultado de um cálculo para a função chamadora. Se a função não usa *return* para retornar um resultado, você deve preceder o nome da função com *void*. O *void* que aparece nos parênteses diz a C que a função não usa parâmetros. Um parâmetro é a informação que o programa passa para a função. Por exemplo, quando seus programas chamam *printf*, as informações que você especifica dentro dos parênteses são *parâmetros*. Quando uma função não usa parâmetros, você deve colocar a palavra *void* dentro dos parênteses. Para usar uma função, você simplesmente especifica o nome da função seguido por parênteses, exatamente como usa *printf*. Os programadores referenciam o uso de uma função como uma *chamada da função*. O programa a seguir, *usa_func.c*, usa a função *oi_pessoal*:

```
#include <stdio.h>

void oi_pessoal(void)
 {
   printf("Oi pessoal!\n");
 }
void main(void)
 {
   oi_pessoal();
 }
```

Ao executar esse programa, a função *main* é executada primeiro. Como você pode ver, o único comando em *main* é a chamada da função *oi_pessoal*. Quando C encontra a chamada da função, imediatamente transfere a execução do programa para a função, iniciando a execução do programa com o primeiro comando na função. Depois que o último comando na função termina, C transfere a execução para o comando que segue imediatamente a chamada da função. Para compreender melhor esse processo, mude a função *main* dentro de *usafunc.c* como mostrado aqui:

```
void main(void)
 {
   printf("Prestes a chamar a função\n");
   oi_pessoal();
   printf("Voltei da chamada da função\n");
 }
```

Quando você compilar e executar o programa *usafunc.c*, sua tela exibirá o seguinte:

```
Prestes a chamar a função
Oi pessoal!
Voltei da chamada da função
C:\>
```

USANDO VARIÁVEIS DENTRO DAS FUNÇÕES 218

À medida que você cria funções úteis, verá que muitas funções requerem que as variáveis gerem resultados valiosos. Para usar uma variável dentro de uma função, você precisa primeiro declarar a variável, exatamente como faz em *main*. Por exemplo, o programa a seguir, *tres_ois.c*, chama a função *tres_olas.c*, que usa a variável *contador* em um laço *for* para exibir uma mensagem três vezes:

```
#include <stdio.h>
```

```
void tres_olas(void)
 {
   int contador; // Variável

   for (contador = 1; contador <= 3; contador++)
     printf("Olá pessoal!\n");
 }

void main(void)
 {
   tres_olas();
 }
```

Quando você declara variáveis dentro de uma função, os nomes usados para essas variáveis são exclusivos para a função. Portanto, se seu programa usa 10 funções diferentes e cada função usa uma variável chamada *contador*, C considera a variável de cada função como distinta. Se sua função requer muitas variáveis, você deverá declarar as variáveis no início da função, exatamente como faria dentro de *main*.

219 COMPREENDENDO MAIN COMO UMA FUNÇÃO

Quando você cria um programa C, usa o nome da função *main* para determinar o primeiro comando que o programa executará. Na verdade, *main* é uma função, de modo que, caso tenha perguntas sobre os tipos de operações que você pode executar dentro de suas funções, a regra é bem simples: *Tudo o que você pode fazer em main, você pode fazer em uma função*. Exatamente como você pode declarar variáveis em *main*, também pode declarar variáveis nas suas funções. Também é possível usar construções tais como *if*, *while* e *for* em suas funções. Finalmente, uma função pode chamar (usar) outra. Por exemplo, o programa a seguir, *chama_2.c*, usa duas funções. Quando o programa inicia, *main* chama a função *tres_olas*, que, por sua vez, chama a função *oi_pessoal* três vezes para exibir mensagens na sua tela, como mostrado aqui:

```
#include <stdio.h>

void ola_pessoal(void)
 {
   printf("Ola, pessoal!\n");
 }

void tres_olas(void)
 {
   int contador;

   for (contador = 1; contador <= 3; contador++)
      ola_pessoal();
 }

void main(void)
 {
   tres_olas();
 }
```

220 INTRODUÇÃO AOS PARÂMETROS

Um *parâmetro* é um valor passado a uma função. A maioria dos programas apresentados neste livro passou parâmetros para a função *printf*, como mostrado aqui:

```
printf("O valor é %d\n", result);
```

À medida que você for usando funções regularmente, poderá passar parâmetros para uma função para melhorar a utilidade da função. Por exemplo, considere a seguinte construção da função *tres_olas*, que chama a função *oi_pessoal* três vezes:

```
void tres_olas(void)
```

```
{
  int contador;

  for (contador = 1; contador <= 3; contador++)
     oi_pessoal();
}
```

Uma função mais útil lhe permite especificar, como um parâmetro, o número de vezes que você quer que o programa exiba a mensagem. Para usar um parâmetro, sua função precisa especificar o nome e o tipo do parâmetro, como mostrado aqui:

```
void ola_conta(int msg_conta)
```

Neste caso, a função *ola_conta* suporta um parâmetro do tipo *int* chamado *msg_conta*. Quando outra função, tal como *main*, quiser usar *ola_conta*, a função precisa especificar o valor que C atribui para o parâmetro *msg_conta*:

```
ola_conta(2);          // Exibe a mensagem duas vezes
ola_conta(100);        // Exibe a mensagem 100 vezes
ola_conta(1);          // Exibe a mensagem uma vez
```

O programa a seguir, *usaparam.c*, ilustra como você poderia usar uma função com um parâmetro:

```
#include <stdio.h>

void oi_pessoal(void)
 {
   printf("Ola, pessoal!\n");
 }

void ola_conta(int msg_conta)
 {
   int contador;

   for (contador = 1; contador <= msg_conta; contador++)
     oi_pessoal();
 }

void main(void)
 {
   printf("Exibe a msg duas vezes\n");
   ola_conta(2);
   printf("Exibe a msg cinco vezes\n");
   ola_conta(5);
 }
```

Como você pode ver, em *main*, a chamada da função para *ola_conta* inclui o valor que C atribuirá ao parâmetro *msg_conta*.

Nota: Ao passar um parâmetro para uma função, o tipo de valor que você passará para o parâmetro (tal como int, float, char etc.) precisa corresponder ao tipo do parâmetro. Dependendo do seu compilador, ele poderá detectar incompatibilidades nos tipos. Se seu compilador não detectar as incompatibilidades nos tipos dos parâmetros, poderão aparecer erros que serão muito difíceis de detectar e de corrigir.

USANDO PARÂMETROS MÚLTIPLOS 221

Como você aprendeu, um *parâmetro* é um valor que você passa para uma função. Em geral, você pode passar um número irrestrito de parâmetros para uma função. No entanto, as pesquisas mostram que, quando o número de parâmetros excede de sete, a função torna-se mais difícil para compreender e usar corretamente, ficando, portanto, mais susceptível a erros. Quando sua função usa mais de um parâmetro, você precisa especificar o tipo e o nome de cada parâmetro e separar os parâmetros por vírgulas, como mostrado aqui:

```
void uma_funcao(int idade, float sal, int num_cargo)
 {
   // Comandos da função
 }
```

Quando seu programa quiser chamar a função, você precisará especificar valores para cada parâmetro, como mostrado aqui:

```
uma_funcao(33, 40000.00, 534);
```

C, por sua vez, atribuirá os valores para os parâmetros, como mostrado na Figura 221:

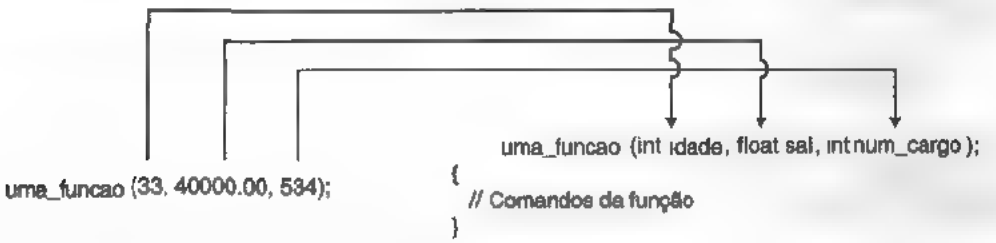

Figura 221 Mapeando os valores dos parâmetros.

222 COMPREENDENDO AS DECLARAÇÕES DE PARÂMETROS EM PROGRAMAS C MAIS ANTIGOS

Quando você cria uma função que usa parâmetros, normalmente especifica o tipo e o nome de cada parâmetro, separados por vírgulas, dentro do cabeçalho da função, como mostrado aqui:

```
void uma_funcao(int idade, float sal, int num_cargo)
 {
    // Comandos da função
 }
```

Se você trabalhar com programas C mais antigos, verá que os programadores declaravam os parâmetros assim:

```
void uma_funcao(idade, sal, num_cargo)
   int idade;
   float sal;
   int num_cargo;
 {
   // Comandos da função
 }
```

Se você encontrar essas declarações de parâmetros, deverá compreender que, embora o formato da declaração seja ligeiramente diferente, o propósito permanece o mesmo — especificar o tipo e o nome do parâmetro. Caso você se sinta tentado a atualizar o formato da função, certifique-se de que seu compilador aceita totalmente o novo formato. Além disso, lembre-se de que, quanto mais modificações você fizer no seu programa, maiores serão suas chances de introduzir um erro. Como regra geral, "*Se está funcionando, não conserte!*"

223 RETORNANDO UM VALOR DE UMA FUNÇÃO

À medida que suas funções se tornarem mais complexas, normalmente realizarão um cálculo e retornarão um resultado. Para fornecer um resultado ao chamador, uma função precisará usar o comando *return*, que você implementará como mostrado aqui:

```
return (resultado);
```

O tipo de valor que a função retorna (*int*, *float*, *char* etc.) determina o tipo da função. Por exemplo, se uma função retorna um valor do tipo *int*, você precisa preceder o nome da função com o nome do tipo, como mostrado aqui:

```
int uma_funcao(int valor)
```

```
{
   // Comandos da função
}
```

A função a seguir, *i_cubo*, retorna o cubo do valor inteiro que o programa especifica como seu parâmetro. Por exemplo, se o chamador passar o valor 5 para a função, *i_cubo* retornará o valor 5 * 5 * 5, ou 125:

```
int      i_cubo (int valor)
{
         return (valor *valor* valor);
{
```

Como você pode ver, a função usa o comando *return* para retornar o resultado do cálculo ao chamador. O *código dentro* da função de chamada pode atribuir o resultado da função chamada (também conhecido como valor de retorno) a uma variável, ou o código pode usar o valor de retorno dentro de uma terceira função, tal como *printf*, como mostrado aqui:

```
result = i_cubo(5);

printf("O cubo de 5 é %d\n", i_cubo(5));
```

O programa a seguir, *i_cubo.c*, usa a função *i_cubo* para determinar vários valores diferentes ao cubo:

```
#include <stdio.h>

int i_cubo(int valor)
{
  return(valor * valor * valor);
}

void main(void)
{
  printf("O cubo de 3 é %d\n", i_cubo(3));
  printf("O cubo de 5 é %d\n", i_cubo(5));
  printf("O cubo de 7 é %d\n", i_cubo(7));
}
```

Os valores que você passa para uma função precisam corresponder aos tipos de parâmetros contidos dentro da declaração dessa função. Por exemplo, se você quiser determinar o cubo de um valor em ponto flutuante, crie uma segunda função chamada *f_cubo*, como mostrado aqui (observe que o valor de retorno também é do tipo *float*):

```
float f_cubo(float valor)
{
  return (valor * valor * valor);
}
```

COMPREENDENDO O COMANDO RETURN 224

Como você aprendeu, para uma função fornecer um resultado ao chamador, ela precisará usar o comando *return*. Quando encontra um comando *return* em uma função, C finaliza imediatamente a execução da função e retorna o valor especificado para o chamador. O programa não executa quaisquer outros comandos dentro da função após o comando *return*. Em vez disso, ele continua a execução na função chamadora.

À medida que você examina outros programas C, pode encontrar funções que contêm múltiplos comandos *return*, cada um dos quais retornando um valor para uma condição específica. Por exemplo, considere a função *compara_valores*, mostrada aqui:

```
int compara_valores(int primeiro, int segundo)
{
  if (primeiro == segundo)
     return(0);
  else if (primeiro > segundo)
     return(1);
```

```
    else if (primeiro < segundo)
       return(2);
 }
```

A função *compara_valores* examina dois valores inteiros e retorna um dos valores listados na Tabela 224.

Tabela 224 Os valores que a função *compara_valores* retorna.

Resultado	Significado
0	Os valores são iguais.
1	O primeiro valor é maior que o segundo.
2	O segundo valor é maior que o primeiro.

Como regra, você deve tentar limitar suas funções a usar somente um comando *return*. À medida que suas funções se tornarem maiores e mais complexas, ter muitos comandos *return* normalmente tornará as funções mais difíceis de compreender. Na maioria dos casos, você pode reescrever sua função para que ela use somente um comando *return*, como mostrado aqui:

```
int compara_valores(int primeiro, int segundo)
 {
   int result;

   if (primeiro == segundo)
      result = 0;
   else if (primeiro > segundo)
      result = 1;
   else if (primeiro < segundo)
      result = 2;

   return(result);
 }
```

Neste caso, como a função é tão simples, você pode ter dificuldade em compreender qual é a vantagem de usar um único comando *return*. No entanto, à medida que suas funções se tornarem mais complexas, a vantagem ficará mais clara. Você deve observar que, algumas vezes, usar mais de um comando *return* produz código mais legível do que a alternativa de um único *return*. Você deverá escrever o código mais legível e facilmente modificável quanto possível; se usar múltiplos comandos *return* atingir seu objetivo, então use tantos comandos *return* quanto forem necessários.

225 COMPREENDENDO OS PROTÓTIPOS DE FUNÇÃO

Se você examinar atentamente o programa anterior, verá que as chamadas das funções sempre aparecem no código do programa-fonte após as funções que elas chamam. A maioria dos novos compiladores C precisa conhecer os tipos de retorno e de parâmetro de uma função antes de o programa chamar a função. Colocando as funções antes de seus chamadores dentro do código do seu programa, você permitirá que o compilador C conheça as informações que ele deverá ter antes de encontrar a chamada da função. No entanto, à medida que seus programas ficam mais complexos, pode tornar-se impossível para você colocar sempre as funções na ordem correta. Portanto, C lhe permite colocar protótipos de função no seu programa, para descrever os tipos de parâmetro e de retorno de uma função. Por exemplo, considere um programa que use as funções *i_cubo* e *f_cubo*, apresentadas na Dica 223. Antes que a função seja usada pela primeira vez, o programa pode incluir um protótipo similar ao seguinte:

```
int i_cubo(int);        // Retorna um parâmetro int
float f_cubo(float);    // Retorna um parâmetro float
```

Como você pode ver, o protótipo da função especifica os tipos de parâmetro e de retorno da função. O programa a seguir, *usaproto.c*, usa dois protótipos de função para eliminar a necessidade de ordem da função.

```
#include <stdio.h>

int i_cubo(int);
float f_cubo(float);
```

```
void main(void)
 {
   printf("O cubo de 3 é %d\n", i_cubo(3));
   printf("O cubo de 3.7 é %f\n", f_cubo(3.7));
 }

int i_cubo(int valor)
 {
   return(valor * valor * valor);
 }

float f_cubo(float valor)
 {
   return(valor * valor * valor);
 }
```

Se você examinar os arquivos de cabeçalho *.h*, tais como *stdio.h*, verá que eles contêm muitos protótipos de funções.

COMPREENDENDO A BIBLIOTECA DE EXECUÇÃO 226

Ao ir escrevendo suas próprias funções, freqüentemente você verá que uma função que criou para um programa atende às necessidades de um segundo programa. A capacidade de reutilizar as funções em mais de um programa pode poupar um tempo considerável de programação e de teste. Na seção Ferramentas da Programação, mais à frente, você aprenderá a colocar suas funções comumente usadas dentro de uma biblioteca para torná-las mais fáceis de usar em múltiplos programas. No entanto, por ora, você pode precisar recortar e colar os comandos da função de um arquivo de código-fonte em outro.

Antes de gastar um tempo enorme desenvolvendo uma ampla variedade de funções de propósito geral, não deixe de examinar as funções que seu compilador fornece. Muitos compiladores referenciam essas funções internas como *biblioteca de execução*. A maioria dos compiladores fornece centenas de funções de biblioteca de execução com propósitos que vão de abertura e trabalho com arquivos para acessar informações do disco ou de diretório para determinar o tamanho de uma string de caracteres. As duas ou três horas que você gastará para ler a documentação da biblioteca de execução pouparão muitas horas de programação que, de outra forma, você gastaria "reinventando a roda".

COMPREENDENDO OS PARÂMETROS FORMAIS E REAIS 227

À medida que você for lendo diferentes livros sobre C, poderá encontrar os termos parâmetros *formais* e *reais*. Em resumo, parâmetros formais são os nomes de parâmetros que aparecem na definição da função. Por exemplo, os nomes *idade*, *salario* e *num_cargo* são os parâmetros normais para a função *info_cargo*, como mostrado aqui:

```
void info_cargo(int idade, float salario, int num_cargo)
 {
   // Comandos da função
 }
```

Quando uma função chama outra função, os valores que a função chamadora passa são os parâmetros reais. No caso da chamada da função a seguir, os valores 30, 4200.00 e 321 sao os parâmetros reais:

```
info_cargo(30, 4200.00, 321);
```

Os parâmetros reais que você passa para uma função podem ser valores constantes ou variáveis. O tipo do valor ou da variável precisa corresponder ao do parâmetro formal. Por exemplo, o fragmento de código a seguir ilustra como usar variáveis como parâmetros reais:

```
int idade_trab = 30;
float salario_trab = 4200.00;
int num_cargo = 321;

info_cargo(idade_trab, salario_trab, num_cargo);
```

Quando você chama uma função com variáveis como os parâmetros reais, os nomes de variável usados para os parâmetros reais não têm relacionamento com os nomes dos parâmetros formais. Em vez disso, C preocupa-se somente com os valores que as variáveis contêm.

228 SOLUCIONANDO OS CONFLITOS NOS NOMES

Como você aprendeu, a maioria dos compiladores C fornece uma extensa biblioteca de funções que você pode chamar para efetuar tarefas específicas. Por exemplo, para obter o valor absoluto de uma expressão inteira, você pode usar a função *abs*. Da mesma forma, para copiar o conteúdo de uma string em outra, você pode usar a função *strcpy*. Ao criar suas próprias funções, algumas vezes uma função que você define terá o mesmo nome que uma função da biblioteca de execução. Por exemplo o programa a seguir, *mstrcpy.c*, cria e usa uma função chamada *strcpy*:

```
#include <stdio.h>

char *strcpy(char *destino, const char *origem)
 {
   char *inicio = destino;

   while (*destino++ = *origem++)
     ;

   return(inicio);
 }

void main(void)
 {
   char titulo[64];

   strcpy(titulo, "Bíblia do Programador C/C++, do Jamsa!");
   printf(titulo);
 }
```

Quando o nome de uma função que você declara dentro de seu programa está em conflito com uma função da biblioteca de execução, C usa a função do seu programa, não a função da biblioteca de execução.

229 FUNÇÕES QUE NÃO RETORNAM INT

Muitas funções vistas anteriormente retornaram valores do tipo *int*. Quando sua função não retorna um valor do tipo *int* (em vez disso, ela pode retornar *float*, *double*, *char* etc.), você precisa informar ao compilador o tipo de retorno da função. O programa a seguir, *exib_med.c*, usa a função *valor_medio* para determinar a média de três valores do tipo *int*. A função retorna a média usando um valor do tipo *float*:

```
#include <stdio.h>

float valor_medio(int a, int b, int c)
 {
   return ((a + b + c) / 3.0);
 }

void main(void)
 {
   printf("A média de 100, 133 e 155 é %f\n", valor_medio(100, 133, 155));
 }
```

Como você pode ver, o cabeçalho da função especifica o tipo de retorno da função:

```
float valor_medio(int a, int b, int c)
```

Nota: Se você não especificar um tipo de retorno da função, o compilador C assumirá que a função retorna o tipo **int**.

COMPREENDENDO AS VARIÁVEIS LOCAIS 230

C lhe permite declarar variáveis dentro de suas funções. Essas variáveis são chamadas *variáveis locais*, pois seus nomes e valores somente têm significado dentro da função que contém a declaração da variável. O programa a seguir, *localerr.c*, ilustra o conceito de uma variável local. A função *valores_locais* declara três variáveis, *a*, *b* e *c*, e atribui às variáveis os valores 1, 2 e 3, respectivamente. A função *main* tenta imprimir o valor de cada variável. No entanto, como os nomes dos valores são locais à função *valores_locais*, o compilador gera erros, dizendo que os símbolos *a*, *b* e *c* estão indefinidos.

```
#include <stdio.h)

void valores_locais(void)
 {
   int a = 1, b = 2, c = 3;

   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }

void main(void)
 {
   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }
```

COMO AS FUNÇÕES USAM A PILHA 231

A seção Gerenciamento da Memória no Windows, mais à frente, descreve a *pilha*, que os programas usam para armazenar informações temporariamente. O propósito principal da pilha é oferecer suporte para as chamadas das funções. Quando seu programa chama uma função, C coloca o endereço da instrução que segue a chamada da função (chamado *endereço de retorno*) na pilha. Em seguida, C coloca os parâmetros da função, da direita para a esquerda, na pilha. Finalmente, se a função declara variáveis locais, C aloca espaço na pilha, que a função pode, então, usar para guardar o valor da variável. A Figura 231 mostra como C usa a pilha para uma única chamada de função.

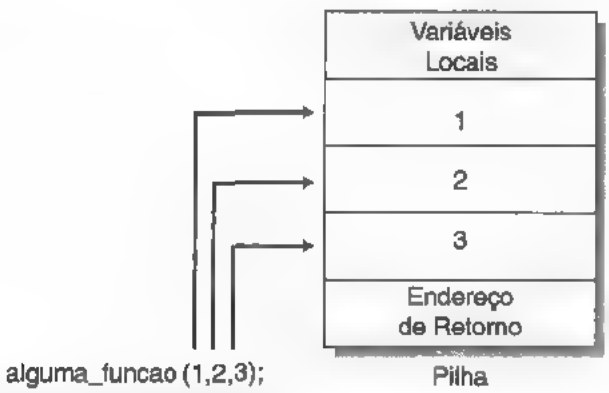

Figura 231 C usa a pilha para a chamada de uma função.

Quando a função termina, C descarta o espaço da pilha que continha as variáveis locais e os parâmetros. Em seguida, C usa o valor de retorno para determinar a instrução que o programa executa em seguida. C remove o valor de retorno da pilha e coloca o endereço no registrador IP (ponteiro da instrução).

COMPREENDENDO A SOBRECARGA DA FUNÇÃO 232

Como você aprendeu na Dica 231, quando seu programa usa uma função, C armazena o endereço de retorno, os parâmetros e as variáveis locais na pilha. Quando a função termina, C descarta o espaço da pilha que continha as variáveis locais e parâmetros, e, depois, usa o valor de retorno para retornar a execução do programa para a posição correta.

Embora o uso da pilha de C seja poderoso porque permite que o programa chame e passe informações para as funções, C também consome tempo de processamento. Os programadores chamam a quantidade de tempo que o computador requer para colocar e retirar informações da pilha de *sobrecarga da função*. Para compreender melhor o impacto da sobrecarga da função no desempenho do seu programa, considere o programa a seguir, *sobrecar.c*. O programa primeiro usa um laço para somar os valores de 1 a 100.000. Em seguida, o programa repete um laço novamente, mas usa uma função para somar os valores, como mostrado aqui:

```
#include <stdio.h>
#include <time.h>

float soma(long int a, float b)
 {
    float result;

    result = a + b;
    return(result);
 }

void main(void)
  {
    long int i;
    float result = 0;
    time_t hora_inicio, hora_parada;

    printf("Trabalhando...\n");
    time(&hora_inicio);
    for (i = 1; i <= 100000L; i++)
      result += i;
    time(&hora_parada);
    printf("Usando laço %d segundos\n", hora_parada -hora_inicio);
    printf("Trabalhando...\n");
    time(&hora_inicio);
    for (i = 1; i <= 100000L; i++)
      result = soma(i, result);
    time(&hora_parada);
    printf("Usando função %d segundos\n", hora_parada -hora_inicio);
 }
```

Na maioria dos sistemas, os cálculos baseados em funções podem requerer quase o dobro do tempo de processamento. Portanto, quando você usar funções dentro de seus programas, precisará considerar os benefícios que elas oferecem (tais como facilidade de uso, reutilização de uma função existente, redução de teste, facilidade de compreensão, e assim por diante) versus a sobrecarga no desempenho que introduzem.

233 COMPREENDENDO ONDE C COLOCA AS VARIÁVEIS LOCAIS

Como foi visto, C lhe permite declarar variáveis dentro de suas funções. Essas variáveis são locais à função, o que significa que somente a função na qual você declarou as variáveis conhece seus valores e existência. A seguinte função, *usa_abc*, declara três variáveis locais chamadas *a*, *b* e *c*:

```
void usa_abc(void)
 {
   int a, b, c;

   a = 3;
   b = a + 1;
   c = a + b;
   printf("a contém %d b contém %d c contém %d\n", a,b,c);
 }
```

Toda vez que seu programa chama a função, C aloca espaço na pilha para armazenar as variáveis locais *a*, *b* e *c*. Quando a função termina, C descarta o espaço anteriormente alocado na pilha e os valores que as variáveis locais continham. Mesmo que sua função declare muitas variáveis locais, C armazena o valor de cada variável na pilha.

DECLARANDO VARIÁVEIS GLOBAIS 234

Na Dica 218 você aprendeu que as variáveis locais são variáveis definidas dentro de uma função cujos nomes e existência são conhecidos somente função. Além das *variáveis locais*, C também permite que seus programas usem variáveis globais, cujos nomes, valores e existência são conhecidos em todo o seu programa. Em outras palavras, todos os programas C podem usar variáveis globais. O programa a seguir, *global.c*, ilustra o uso de três variáveis globais, *a*, *b* e *c*:

```
#include <stdio.h>
int a = 1, b = 2, c = 3;  // Variáveis globais

void valores_globais(void)
 {
   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }

void main(void)
 {
   valores_globais();
   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }
```

Quando você compila e executa este programa, as funções *valores_globais* e *main* exibem os valores da variável global. Observe que você declara as variáveis fora de todas as funções. Ao declarar variáveis globais deste modo, todas as funções do seu programa podem usar e alterar os valores da variável global simplesmente referenciando o nome da variável global. Embora as variáveis globais possam parecer convenientes, o uso incorreto delas pode causar erros que são difíceis de depurar, como você aprenderá na dica a seguir.

EVITE USAR VARIÁVEIS GLOBAIS 235

Na dica anterior vimos como declarar variáveis globais, que seu programa conhece em todas as suas funções. À primeira vista, usar variáveis globais parece simplificar a programação porque elimina a necessidade de parâmetros de funções e, mais importante, a necessidade de compreender a *chamada por valor* e a *chamada por referência*. No entanto, infelizmente, as variáveis globais com freqüência criam mais erros do que corrigem. Como seu código pode mudar o valor de uma variável global em virtualmente qualquer ponto dentro do seu programa, é muito difícil para outro programador que esteja lendo seu programa encontrar cada local no programa onde a variável global é alterada. Portanto, outros programadores podem fazer mudanças no seu programa sem compreender totalmente o efeito que a modificação tem em uma variável global. Como regra, as funções somente devem modificar aquelas variáveis passadas para as funções como parâmetros. Isso permite que os programadores estudem os protótipos da função para determinar rapidamente quais variáveis uma função altera.

SOLUCIONANDO OS CONFLITOS DE NOMES DE VARIÁVEIS LOCAIS E GLOBAIS 236

Como você aprendeu, as variáveis locais são variáveis que você declara dentro de uma função cujos nomes são conhecidos somente para essa função. Por outro lado, quando você declara variáveis globais fora de todas as funções, toda função em todo o seu programa conhecerá os nomes delas. Se seu programa usa variáveis globais, algumas vezes o nome de uma variável global é o mesmo que aquele de uma variável local que seu programa declara dentro de uma função. Por exemplo, o programa a seguir, *conflito.c*, usa as variáveis globais *a*, *b* e *c*. A função *conflito_a* usa uma variável local chamada *a* e as variáveis globais *b* e *c*.

```
#include <stdio.h>
int a = 1, b = 2, c = 3;        // Variáveis globais

void conflito_a(void)
 {
   int a = 100;

   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }

void main(void)
 {
   conflito_a();
   printf("a contém %d b contém %d c contém %d\n", a, b, c);
 }
```

Quando você compilar e executar o programa *conflito.c*, sua tela exibirá o seguinte:

```
a contém 100 b contém 2 c contém 3
a contém 1 b contém 2 c contém 3
C:\>
```

Quando nomes de variáveis globais e nomes de variáveis locais estão em conflito, C sempre usará a variável local. Como você pode ver, as alterações que a função *conflito_a* fez na variável *a* somente aparecem dentro da função.

Nota: Embora o propósito deste programa seja ilustrar como C soluciona os conflitos de nomes, ele também ilustra a confusão que pode ocorrer quando você usa variáveis globais. Neste caso, um programador que esteja lendo seu código precisa prestar muita atenção para determinar que a função não altere a variável global a, mas, sim, uma variável local. Como a função combina o uso de variáveis globais e locais, o código pode tornar-se difícil de entender.

237 DEFININDO MELHOR O ESCOPO DE UMA VARIÁVEL GLOBAL

Vimos na Dica 234 que uma variável global é uma variável que todas as funções no seu programa conhecem. Dependendo de onde você define uma variável global, é possível controlar quais funções são na realidade capazes de referenciar a variável. Em outras palavras, você pode controlar o *escopo* da variável global. Quando seu programa declara uma variável global, quaisquer funções que seguem a declaração da variável podem referenciar essa variável, até o final do arquivo-fonte. As funções que têm definições que aparecem antes da definição da variável global não podem acessar a variável global. Como um exemplo, considere o programa a seguir, *escopogl.c*, que define a variável global *titulo*:

```
#include <stdio.h>

void titulo_desconhecido(void)
 {
   printf("O título do livro é %s\n", titulo);
 }

char titulo[]="Bíblia do Programador C/C++, do Jamsa!";

void main(void)
 {
   printf("Titulo: %s\n", titulo);
 }
```

Como você pode ver, a função *titulo_desconhecido* tentará exibir a variável *titulo*. No entanto, como a declaração da variável global ocorre após a definição da função, a variável global é desconhecida dentro da função. Quando você tentar compilar este programa, seu compilador gerará um erro. Para corrigir o erro, coloque a declaração da variável global antes da função.

COMPREENDENDO A CHAMADA POR VALOR 238

Já sabemos que seus programas passam informações para as funções usando parâmetros. Quando você passa um parâmetro para uma função, C usa uma técnica conhecida como *chamada por valor* para fornecer à função uma cópia dos valores dos parâmetros. Usando a chamada por valor, quaisquer modificações que a função fizer nos parâmetros existem somente dentro da própria função. Quando a função termina, o valor das variáveis que a função chamadora passou para a função não é modificado dentro da função chamadora. Por exemplo, o programa a seguir, *naomuda.c*, passa três parâmetros (as variáveis *a*, *b* e *c*) para a função *exibe_e_altera*. A função, por sua vez, exibirá os valores, somará 100 aos valores e depois exibirá o resultado. Quando a função terminar, o programa exibirá os valores das variáveis. Como C usa chamada por valor, a função não altera os valores das variáveis dentro do chamador, como mostrado aqui:

```
#include <stdio.h>

void exibe_e_altera(int primeiro, int segundo, int terceiro)
 {
   printf("Valores originais da função %d %d %d\n", primeiro, segundo,
    terceiro);
   primeiro += 100;
   segundo += 100;
   terceiro += 100;
   printf("Valores finais da função %d %d %d\n", primeiro, segundo, terceiro);
 }

void main(void)
 {
   int a = 1, b = 2, c = 3;

   exibe_e_altera(a, b, c);
   printf("Valores finais em main %d %d %d\n", a,b,c);
 }
```

Quando você compilar e executar o programa *naomuda.c*, sua tela exibirá o seguinte:

```
Valores originais da função 1 2 3
Valores finais da função 101 102 103
Valores finais em main 1 2 3
C:\>
```

Como você pode ver, as alterações que a função faz nas variáveis somente são visíveis dentro da própria função. Quando a função termina, as variáveis dentro de *main* estão inalteradas.

Nota: Quando você usa chamada por referência (que será apresentada em detalhe na Dica 240), a função pode modificar o valor de um parâmetro para que a modificação seja visível fora da função.

EVITANDO A ALTERAÇÃO NO VALOR DO PARÂMETRO COM A CHAMADA POR VALOR 239

Na Dica 238 você aprendeu que, por padrão, C usa chamada por valor para passar parâmetros para as funções. Consequentemente, quaisquer alteraçoes nos valores dos parâmetros ocorrem apenas dentro da própria função. Quando a função termina, os valores das variáveis que o programa passou para a função estão inalterados. Como detalha a seção "Introdução à Linguagem C", uma variável é basicamente um nome atribuído a uma posição de memória. Toda variável tem dois atributos de interesse — seu valor atual e seu endereço de memória. No caso

do programa *naomuda.c*, apresentado na dica anterior, as variáveis *a*, *b* e *c* poderiam usar os endereços de memória mostrados na Figura 239.1:

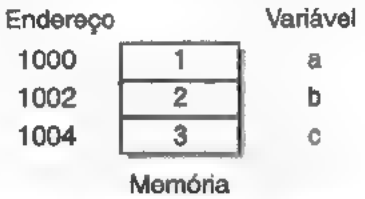

Figura 239.1 As variáveis armazenam um valor e residem em uma posição de memória específica.

Quando você passa parâmetros para uma função, C coloca os valores correspondentes na pilha. No caso das variáveis *a*, *b* e *c*, a pilha contém os valores 1, 2 e 3. Quando a função acessa os valores da variável, a função referencia as posições da pilha, como mostrado na Figura 239.2.

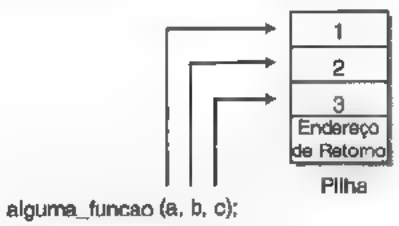

Figura 239.2 As funções referenciam valores armazenados na pilha.

Quaisquer modificações que a função fizer nos valores dos parâmetros realmente alteram os valores da pilha, como mostrado na Figura 239.3:

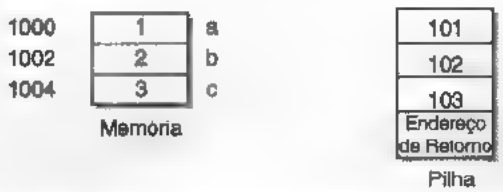

Figura 239.3 As modificações que as funções fazem nos valores dos parâmetros afetam apenas os valores que estão na pilha.

Quando a função termina, C descarta os valores na pilha bem como as alterações que a função fez nos conteúdos da pilha. A função nunca referencia as posições de memória que contêm o valor de cada variável, de modo que suas funções não podem fazer alterações que existam após a função terminar em qualquer parâmetro que a *função recebe usando a chamada por valor.*

240 COMPREENDENDO A CHAMADA POR REFERÊNCIA

Como você aprendeu, C passa parâmetros para as funções usando chamada por valor por padrão. Usando a chamada por valor, as funções não podem modificar o valor de uma variável passada para uma função. No entanto, na maioria dos programas, suas funções modificarão as variáveis de um modo ou de outro. Por exemplo, uma função que lê informações de um arquivo precisa colocar as informações em uma matriz de strings de caracteres. Da mesma forma, uma função tal como *strupr* (apresentada na seção Strings deste livro) precisa converter as letras em uma string de caractere para maiúsculas. Quando suas funções alteram o valor de um parâmetro, seus programas precisam passar o parâmetro para a função usando *chamada por referência*. A diferença entre chamada por valor e chamada por referência é que, usando a chamada por valor, as funçoes recebem uma cópia do valor de um parâmetro. Por outro lado, com a chamada por referência, as funções recebem o endereço de memória da variável. Portanto, as funções podem alterar o valor armazenado na posição de memória específica (em outras palavras, o valor da variável); alterações essas que permanecem após a função terminar. Para usar a chamada por referência, seu programa precisa usar *ponteiros*. A seção Matrizes, Ponteiros e Estruturas mais à frente discute os ponteiros em detalhes. No entanto, por ora, pense em um ponteiro simplesmente como um endereço de memó-

ria. Para atribuir o endereço de uma variável a um ponteiro, use o operador de endereço de C (&). Para acessar posteriormente o valor na posição de memória para a qual o ponteiro aponta, use o operador de redireção (*). As Dicas 241 e 242 discutem esses operadores em detalhes.

OBTENDO UM ENDEREÇO 241

Uma variável é essencialmente um nome atribuído a uma ou mais posições de memória. Quando seu programa roda, cada variável reside em seu próprio endereço de memória. Seu programa localiza as variáveis na memória usando o *endereço de memória* da variável. Para determinar o endereço de uma variável, use o operador de endereço (&). Por exemplo, o programa a seguir, *exib_end.c*, usa o operador de endereço para exibir o endereço (em hexadecimal) das variáveis *a*, *b* e *c*:

```
#include <stdio.h>
void main(void)
 {
   int a = 1, b = 2, c = 3;

   printf("O endereço de a é %x o valor de a é %d\n", &a, a);
   printf("O endereço de b é %x o valor de b é %d\n", &b, b);
   printf("O endereço de c é %x o valor de c é %d\n", &c, c);
 }
```

Quando você compilar e executar este programa, ele exibirá uma saída similar a esta (os valores dos endereços reais serão diferentes):

```
O endereço de a é fff4 o valor de a é 1
O endereço de b é fff2 o valor de b é 2
O endereço de c é fff0 o valor de c é 3
C:\>
```

Quando seus programas mais tarde passarem parâmetros para funções cujos valores a função precisa alterar, seus programas passarão as variáveis por referência (endereço de memória), usando o operador de endereço, como mostrado aqui:

```
alguma_funcao(&a, &b, &c);
```

USANDO UM ENDEREÇO DE MEMÓRIA 242

Na Dica 241 você aprendeu como usar o operador de endereço de C para obter o endereço de memória de uma variável. Quando você passa um endereço para uma função, precisa dizer ao compilador C que a função estará usando um ponteiro (o endereço de memória) de uma variável, e não o valor da variável. Para fazer isso, você precisa declarar uma *variável ponteiro*. Declarar uma variável ponteiro é muito similar à declaração de uma variável padrão, em que você especifica um tipo e o nome da variável. A diferença, entretanto, é que um asterisco (*) precede os nomes das variáveis ponteiro. As declarações a seguir criam variáveis ponteiro do tipo *int*, *float* e *char*.

```
int *i_ponteiro;
float *f_ponteiro;
char *c_ponteiro;
```

Após você declarar uma variável ponteiro, precisa atribuir um endereço de memória a ela. Por exemplo, o comando a seguir, atribui o endereço da variável inteira *a* à variável ponteiro *i_ponteiro*:

```
i_ponteiro = &a;
```

Em seguida, para usar o valor apontado pela variável ponteiro, seus programas precisam usar o operador de redireção de C — o asterisco (*). Por exemplo, o comando a seguir atribui o valor 5 à variável *a* (cujo endereço está contido na variável *i_ponteiro*):

```
*i_ponteiro = 5;
```

De um modo similar, o comando a seguir atribui à variável *b* o valor ao qual a variável *i_ponteiro* aponta atualmente:

```
b = *i_ponteiro;
```

Quando você quiser usar o valor apontado por uma variável ponteiro, use o operador de redireção (*). Quando quiser atribuir o endereço de uma variável a uma variável ponteiro, use o operador de endereço (&). O programa a seguir, *usa_end.c*, ilustra o uso de uma variável ponteiro. O programa atribui à variável ponteiro *i_ponteiro* o endereço da variável *a*. O programa então usa a variável ponteiro para alterar, exibir e atribuir o valor da variável:

```
#include <stdio.h>

void main(void)
 {
   int a = 1, b = 2;
   int *i_ponteiro;

   i_ponteiro = &a;  // Atribui um endereço
   *i_ponteiro = 5;   // Altera o valor apontado por i_ponteiro para 5

   // Exibe o valor
   printf("O valor apontado por i_ponteiro é %d a variável a é %d\n",
   *i_ponteiro, a);
   b = *i_ponteiro;  // Atribui o valor
   printf("O valor de b é %d\n", b);
   printf("Valor de i_ponteiro %x\n", i_ponteiro);
 }
```

Lembre-se de que um ponteiro não é nada mais que um endereço de memória. Seu programa precisa atribuir o valor que o ponteiro (o endereço) contém. No programa *usa_end.c*, o programa atribuiu ao ponteiro o endereço da variável *a*. O programa poderia ter também atribuído o endereço da variável *b*.

Nota: Quando você usa ponteiros, precisa ainda ter em mente os tipos de valores, tais como int, float e char. Seus programas somente devem atribuir o endereço de valores inteiros às variáveis ponteiro, e assim por diante.

243 ALTERANDO O VALOR DE UM PARÂMETRO

Como você aprendeu, para alterar o valor de um parâmetro dentro de uma função, seus programas precisam usar a chamada por referência, passando o endereço da variável. Dentro da função, você precisa usar ponteiros. O programa a seguir, *altparam.c*, usa ponteiros e endereços (chamada por referência) para exibir e, depois, alterar os parâmetros que o programa passa para a função *exibe_e_altera:*

```
#include <stdio.h>

void exibe_e_altera(int *primeiro, int *segundo,
                    int *terceiro)
 {
   printf("Valores originais da função %d %d %d\n", *primeiro, *segundo,
   *terceiro);
   *primeiro += 100;
   *segundo += 100;
   *terceiro += 100;

   printf("Valores finais da função %d %d %d\n", *primeiro, *segundo,
   *terceiro);
 }

void main(void)
```

```
{
  int a = 1, b = 2, c = 3;

  exibe_e_altera(&a, &b, &c);
  printf("Valores finais em main %d %d %d\n", a, b ,c);
}
```

Já vimos que, quando o programa chama a função, ele passa como parâmetros os endereços das variáveis *a*, *b* e *c*. Dentro de *exibe_e_altera*, a função usa variáveis ponteiro e o operador de redireção para alterar e exibir os valores dos parâmetros. Quando você compilar e executar o programa *altparam.c*, sua tela exibirá a seguinte saída:

```
Valores originais da função 1 2 3
Valores finais da função 101 102 103
Valores finais em main 101 102 103
C:\>
```

ALTERANDO SOMENTE PARÂMETROS ESPECÍFICOS 244

Como você aprendeu, suas funções podem modificar o valor de um parâmetro usando a chamada por referência. Por exemplo, a dica anterior, apresentou a função *exibe_e_altera*, que usou chamada por referência para alterar o valor de cada um de seus parâmetros. No entanto, em muitos casos, suas funções podem alterar o valor de um parâmetro e ao mesmo tempo deixar o valor de um segundo parâmetro inalterado. Por exemplo, o programa a seguir, *mudaprim.c*, usa a função *muda_primeiro* para atribuir ao parâmetro *primeiro* o valor do parâmetro *segundo*:

```
#include <stdio.h>

void muda_primeiro(int *primeiro, int segundo)
 {
   *primeiro = segundo; // Atribui o valor de segundo a primeiro
 }

void main(void)
 {
   int a = 0, b = 5;

   muda_primeiro(&a, b);
   printf("Valor de a %d   valor de b %d\n", a, b);
 }
```

Como você pode ver, a função *muda_primeiro* usa a chamada por referência para alterar o valor do parâmetro *primeiro*, e chamada por valor para o parâmetro *segundo*. Quando seus programas usam ambas as técnicas — e eles usarão —, você precisa ter em mente quando usar ponteiros e quando referenciar diretamente a variável. Como regra, os parâmetros cujos valores você quer alterar irão requerer chamada por referência. Para compreender melhor o impacto da chamada por referência *versus* chamada por valor, modifique a função *muda_primeiro*, como mostrado aqui:

```
void muda_primeiro(int *primeiro, int segundo)
 {
   *primeiro = segundo; Atribui o valor de segundo a primeiro
   segundo = 100;
 }
```

Quando você compilar e executar este programa, verá que o valor de *primeiro* foi alterado, mas o valor de *segundo* não foi. Como o parâmetro *segundo* foi passado usando chamada por valor, a alteração do parâmetro não é visível fora da função.

245 A CHAMADA POR REFERÊNCIA AINDA USA A PILHA

Já vimos que, ao passar parâmetros para as funções, C coloca os valores dos parâmetros na pilha. C usa a pilha para armazenar os parâmetros esteja você usando chamada por valor ou chamada por referência. Quando você passa um parâmetro por *valor*, C coloca o valor do parâmetro na pilha. Ao passar um parâmetro por referência, C coloca o *endereço* do parâmetro na pilha. A Dica 244 apresentou o programa *mudaprim.c*, que usou a função *muda_primeiro* para atribuir o valor do parâmetro segundo da função para o parâmetro primeiro da função. Quando o programa chama a função, C coloca o endereço da variável *a* e o valor da variável *b* na pilha, como mostrado na Figura 245.

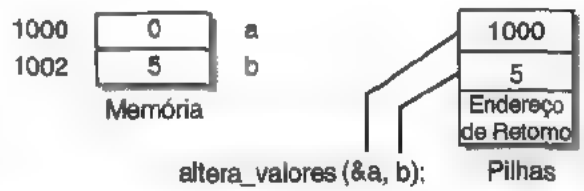

Figura 245 C coloca um endereço e um valor na pilha.

Como a função *altera-valores* referencia a posição de memória que contém o valor da variável *a*, as alterações que *altera-valores* faz na variável existem após a função terminar.

246 INTRODUZINDO AS VARIÁVEIS DE FUNÇÕES QUE LEMBRAM

Em C, as variáveis que você declara dentro de funções são freqüentemente chamadas *automáticas*, porque são criadas automaticamente pelo compilador quando a função inicia, sendo, depois, destruídas quando a função termina. A vida automática da variável ocorre porque o compilador armazena variáveis das funções temporariamente na pilha. Como resultado, se uma função atribuir um valor a uma variável durante uma chamada, a variável perderá seu valor quando a função terminar. A próxima vez que você chamar a função, o valor da variável estará novamente indefinido. Dependendo do processamento que sua função fizer, algumas vezes você poderá querer que as variáveis da função memorizem o último valor que receberam dentro da função.

Por exemplo, assuma que você tenha escrito uma função chamada *imprime_boletim*, que imprime um boletim para todo aluno em uma escola. Sua função poderia usar a variável *id_aluno* para armazenar o número de identificação do último aluno que teve seu boletim impresso. Desse modo, sem ser instruída, a função pode iniciar com o próximo aluno. Para fazer as variáveis locais da sua função memorizar seus valores desse modo, você precisa declarar as variáveis usando a palavra-chave *static*, como mostrado aqui:

```
void imprime_boletim(int num_impressora)
 {
   static int id_aluno;

   // Outros comandos
 }
```

O programa a seguir, *estatica.c*, ilustra o uso de uma variável estática dentro de uma função. O programa, que usa a função *imprime_boletim*, inicia atribuindo à variável *id_aluno* o valor 100. Cada vez que o programa chamar, a função, a função exibirá o valor da variável, e, depois, incrementará o valor por 1, como mostrado aqui:

```
#include <stdio.h>
void imprime_boletim(int num_impressora)
 {
   static int id_aluno = 100;

   printf("Imprimindo boletim para o aluno %d\n",id_aluno);  id_aluno++;
   // Outros comandos aqui
 }

void main(void)
 {
```

```
    imprime_boletim(1);
    imprime_boletim(1);
    imprime_boletim(1);
}
```

Quando você compilar e executar o programa *estatica.c*, sua tela exibirá a seguinte saída:

```
Imprimindo boletim para o aluno 100
Imprimindo boletim para o aluno 101
Imprimindo boletim para o aluno 102
C:\>
```

Como você pode ver, a variável *id_aluno* retém seu valor de uma chamada para a próxima.

Nota: Quando você declara variáveis estáticas, o compilador C não armazena as variáveis na pilha. Em vez disso, ele coloca as variáveis dentro do segmento de dados para que seus valores continuem existindo.

COMPREENDENDO COMO C INICIALIZA VARIÁVEIS ESTÁTICAS 247

Na dica anterior você aprendeu que a palavra-chave *static* instrui o compilador a reter o valor de uma variável da chamada de uma função para a próxima chamada. Quando sua função declara uma variável estática, C permite-lhe inicializar a variável, como mostrado aqui:

```
void imprime_boletim(int num_impressora)
{
  static int id_aluno = 100;  // Inicializada uma vez

  // Outros comandos
}
```

Quando você declara uma variável como *estática*, o compilador inicializará a variável com o valor que você especificar. Quando você chamar a função mais tarde, C *não efetuará* a atribuição de inicialização novamente. Essa inicialização de variável na função é diferente do processamento que C normalmente executa dentro de uma função. No caso da função a seguir, C inicializará a variável *conta* toda vez que o programa chamar a função:

```
void alguma_funcao(int idade, char *nome)
{
  int conta = 1;   // Inicializada em toda chamada

  // Outros comandos
}
```

USANDO A SEQÜÊNCIA DE CHAMADA PASCAL 248

À medida que você vai criando programas C, poderá descobrir que gostaria de usar uma função que criou anteriormente em Pascal. Dependendo do seu compilador, linkeditor e tipo de biblioteca, poderá ainda ser possível chamar a função Pascal a partir do seu programa C. Os passos que você precisará executar para fazer isso, no entanto, dependerão do seu compilador. Adicionalmente, dentro do código do seu programa, você precisará incluir um protótipo de função no início do seu programa que inclui a palavra-chave *pascal*, como mostrado aqui:

```
int pascal alguma_funcao(int nota, int nivel);
```

Se você programar no ambiente Windows, verá que muitas funções da biblioteca de execução usam a seqüência de chamada Pascal. As funções que usam a palavra-chave *pascal* não podem suportar um número variável de argumentos (ao contrário de *printf* e *scanf*).

COMPREENDENDO O EFEITO DA PALAVRA-CHAVE PASCAL 249

A dica anterior mostrou que, quando seus programas chamam uma função, C passa parâmetros para a função usando a pilha. C coloca parâmetros na pilha da direita para a esquerda. A Figura 249.1 ilustra o conteúdo da pilha para uma chamada de função em C.

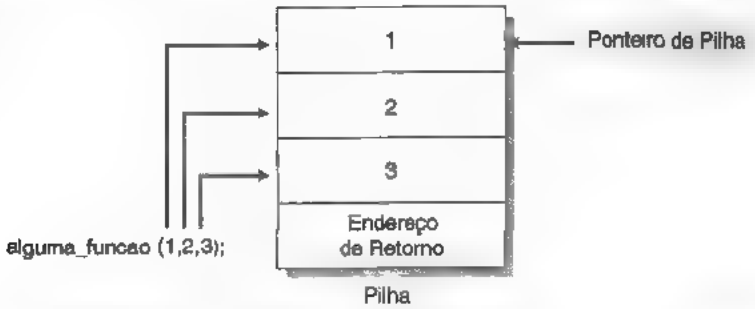

Figura 249.1 O conteúdo da pilha para a chamada de uma função em C.

Por outro lado, Pascal coloca argumentos na pilha da esquerda para a direita. A Figura 249.2 ilustra o conteúdo da pilha para a chamada de função Pascal.

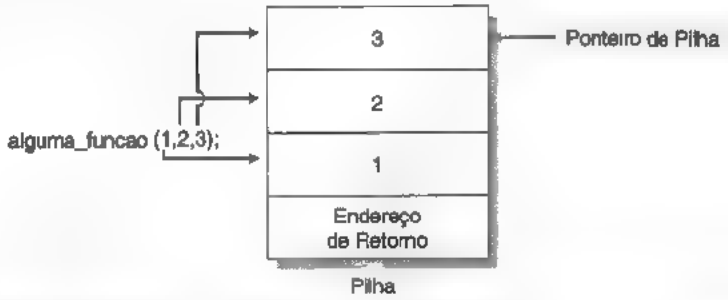

Figura 249.2 O conteúdo da pilha para a chamada de uma função Pascal.

Se você estiver usando uma função Pascal dentro de seu programa C, use a palavra-chave *pascal* para instruir o compilador C a colocar os parâmetros na pilha da esquerda para a direita, na ordem em que Pascal espera.

250 ESCREVENDO UM EXEMPLO DE LINGUAGEM MISTA

Como você aprendeu, muitos compiladores C lhe permitem chamar funções que foram escritas em uma linguagem de programação diferente. Se você estiver chamando uma função Pascal de dentro de seu programa C, por exemplo, pode preceder o protótipo da função usando a palavra-chave *pascal*. Já falamos que a palavra-chave *pascal* instrui o compilador a colocar os parâmetros na pilha da esquerda para a direita. Para ilustrar o processamento que a palavra-chave *pascal* efetua, crie a seguinte função, *exibe_valores* e preceda a função com a palavra-chave *pascal*:

```
#include <stdio.h>

void pascal exibe_valores(int a, int b, int c)
 {
   printf("a %d  b %d  c %d\n", a, b, c);
 }
```

Em seguida, chame a função usando o seguinte código de programa:

```
void main(void)
 {
   exibe_valores(1, 2, 3);
   exibe_valores(100, 200, 300);
 }
```

Para experimentar a função *exibe_valores*, remova a palavra-chave *pascal* e observe a mudança na ordem dos valores dos parâmetros que C exibe. Se seus programas posteriormente chamarem uma rotina Pascal, você precisará usar a palavra-chave *pascal* no protótipo da função.

COMPREENDENDO A PALAVRA-CHAVE CDECL 251

Na Dica 250 você aprendeu que, se usar funções escritas em Pascal, usará a palavra-chave *pascal* para instruir o compilador a colocar os parâmetros na pilha na ordem correta. Quando você usar funções escritas com múltiplas linguagens de programação, provavelmente irá querer incluir a palavra-chave *cdecl* dentro dos protótipos de suas funções para indicar as funções em C e para oferecer clareza ao leitor. Por exemplo, o seguinte protótipo de função informa o compilador de que a função *muda_valores* usa a estrutura de chamada de C:

```
int cdecl muda_valores(int *, int *, int *);
```

Quando o compilador encontra a palavra-chave *cdecl* dentro do cabeçalho de uma função, ele faz os parâmetros passados para a função serem colocados na pilha da direita para a esquerda. Além disso, o compilador garantirá que o linkeditor use o formato C para o nome da função.

COMPREENDENDO A RECURSÃO 252

Como você viu, C lhe permite dividir seu programa em partes menores chamadas *funções*. Usando funções, o programa torna-se mais fácil de compreender, programar e testar. Além disso, você normalmente poderá usar as funções que criar para um programa dentro de outro programa. À medida que seus programas são executados, uma função pode chamar outra, que chama outra, que pode, por sua vez, chamar várias outras funções. Dentro da série, cada função executa uma operação específica. C permite até mesmo que uma função chame a si mesma! Uma *função recursiva* é uma função que chama a si mesma para executar uma operação específica. O processo de uma função chamar a si mesma é conhecido como *recursão*. À medida que a complexidade de seus programas e funções aumentar, você descobrirá que pode facilmente definir muitas operações em termos recursivos.

Quando você usa programas e funções complexas, pode querer criar uma função recursiva. Por exemplo, muitos livros sobre programação usam o problema do fatorial para ilustrar como a recursão funciona. O fatorial de 1 é 1. O fatorial de 2 é 2*1. O fatorial de 3 é 3*2*1. Da mesma forma, o fatorial de 4 é 4*3*2*1. O processo do fatorial pode prosseguir infinitamente. Se você examinar o processamento que o fatorial realiza, verá que o fatorial de 4, por exemplo, é na verdade 4 vezes o fatorial de 3 (3*2*1). Da mesma forma, o fatorial de 3 é, na verdade, 3 vezes o fatorial de 2 (2*1). O fatorial de 2 é 2 vezes o fatorial de 1 (1). A Tabela 252 ilustra o processamento do fatorial.

Tabela 252 Processamento do fatorial.

Valor	Cálculo	Resultado	Fatorial
1	1	1	1
2	2*1	2	2 * Fatorial(1)
3	3*2*1	6	3 * Fatorial(2)
4	4*3*2*1	24	4 * Fatorial(3)
5	5*4*3*2*1	120	5 * Fatorial(4)

O programa a seguir, *fator.c*, cria a função recursiva *fatorial*, e, depois, usa a função para retornar os valores de fatorial para os valores de 1 até 5:

```
#include <stdio.h>

int fatorial(int valor)
 {
   if (valor == 1)
      return(1);
   else
      return(valor * fatorial(valor-1));
 }

void main(void)
 {
   int i;

   for (i = 1; i <= 5; i++)
```

```
    printf("O fatorial de %d é %d\n", i, fatorial(i));
}
```

Como você pode ver, a função *fatorial* retorna um resultado que está baseado no resultado da própria função. A dica a seguir examinará a função *fatorial* em detalhes.

253 COMPREENDENDO A FUNÇÃO RECURSIVA FATORIAL

Na Dica 252 você aprendeu que uma função recursiva é uma função que chama a si mesma para efetuar uma tarefa específica, apresentando a função *fatorial* para ilustrar a recursão. A função *fatorial* recebe um valor de parâmetro específico. Quando a função inicia, ela primeiro verifica se o valor é 1, que, pela definição de fatorial, é 1. Se o valor for 1, a função retornará o valor 1. Se o valor não for 1, a função retornará o resultado do valor vezes o fatorial do valor menos 1.

Por exemplo, suponha que o programa chame a função com o valor 3. A função retornará o resultado de *3 * fatorial(3-1)*. Quando C encontrar a chamada da função dentro do comando *return*, C chamará a função uma segunda vez — desta vez com valor de 3-1 ou 2. Novamente, como o valor não é 1, a função retorna o resultado de *2 * fatorial(2-1)*. Na terceira vez em que a função é chamada, o valor é 1. Como resultado, a função retorna o valor para a função chamadora, o que, por sua vez, retorna o resultado de 2*1 para a função chamadora. A função chamadora então retorna o resultado de 3*2*1 para *sua* função chamadora. A Figura 253 ilustra a cadeia de chamadas de funções recursivas e valores de retorno para a chamada da função fatorial(3).

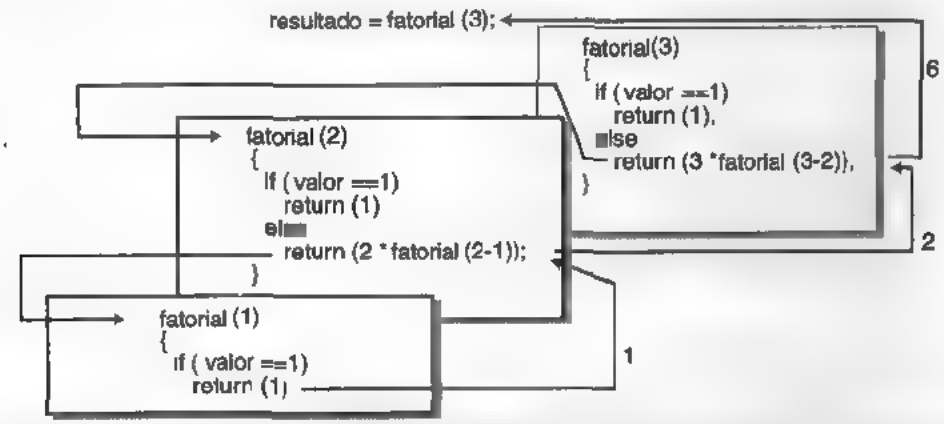

Figura 253 O encadeamento de chamadas de função e valores de retorno para a função recursiva *fatorial*.

Uma função recursiva é similar a uma construção de repetição, em que você precisa especificar a condição de término. Se você não especificar uma condição de término, a função nunca terá fim. No problema do fatorial, a condição de término é o fatorial de 1, que é, por definição, 1.

254 PROGRAMANDO OUTRO EXEMPLO RECURSIVO

Na Dica 252 você aprendeu que uma função recursiva é uma função que chama a si mesma para efetuar processamento específico. A dica anterior, por sua vez, apresentou e explicou a função recursiva *fatorial*. Como a recursão pode ser um conceito difícil, esta dica apresenta mais uma função recursiva, *exibe_invert*, que exibirá as letras de uma string em ordem invertida. Dadas as letras ABCDE, a função exibirá na tela EDCBA. O programa a seguir, *inverte.c*, usa a função *exibe_invert*:

```
#include <stdio.h>

void exibe_invert(char *string)
 {
   if (*string)
    {
       exibe_invert(string+1);
```

```
        putchar(*string);
    }
}

void main(void)
{
  exibe_invert("ABCDE");
}
```

EXIBINDO VALORES PARA COMPREENDER MELHOR A RECURSÃO 255

Já foi visto que uma função recursiva é uma função que chama a si mesma para efetuar uma operação específica. A Dica 252 apresentou a função recursiva *fatorial*. Para lhe ajudar a compreender melhor o processo de recursão, o programa *exibefat.c* inclui comandos *printf* dentro da função *fatorial* para ilustrar o processamento recursivo da função dentro do programa:

```
#include <stdio.h>

int fatorial(int valor)
{
  printf("Em fatorial com o valor %d\n", valor);
  if (valor == 1)
  {
    printf("Retornando o valor 1\n");
    return(1);
  }
  else
  {
    printf("Retornando %d * fatorial(%d)\n", valor, valor-1);
    return(valor * fatorial(valor-1));
  }
}

void main(void)
{
  printf("O fatorial de 4 é %d\n", fatorial(4));
}
```

Quando você compilar e executar o programa *exibefat.c*, sua tela exibirá a seguinte saída:

```
Em fatorial com o valor 4
Retornando 4 * fatorial(3)
Em fatorial com o valor 3
Retornando 3 * fatorial(2)
Em fatorial com o valor 2
Retornando 2 * fatorial(1)
Em fatorial com o valor 1
Retornando o valor 1
O fatorial de 4 é 24
C:\>
```

Inserir o comando *printf* em todas as suas funções recursivas lhe ajudará a compreender o processamento que as funções executam.

COMPREENDENDO A RECURSÃO DIRETA E INDIRETA 256

Uma função recursiva é uma função que chama a si mesma para realizar uma certa operação. Várias dicas anteriores nesta seção apresentaram funções recursivas. Quando uma função chama a si mesma para realizar uma

tarefa, a função executa uma *recursão direta*. Após você ter examinado algumas funções recursivas, deve poder compreender a maioria das funções que usa a recursão direta. Uma forma de recursão mais difícil, a *recursão indireta*, ocorre quando uma função (a função A) chama outra função (a função B), que, por sua vez, chama a função original (função A). Como a recursão indireta pode resultar em código que é muito difícil de compreender, como regra você deverá evitar usar a recursão indireta sempre que possível.

257 DECIDINDO USAR OU NÃO A RECURSÃO

Uma função recursiva é uma função que chama a si mesma para realizar uma tarefa específica. Ao criar funções, usando recursão, você pode criar soluções elegantes para muitos problemas. No entanto, você deve evitar a recursão sempre que possível por duas razões. Primeiro, as funções recursivas podem ser difíceis de compreender para os programadores novatos. Segundo, como regra, as funções recursivas normalmente são consideravelmente mais lentas que suas correspondentes não-recursivas. O programa a seguir, *sem_rec.c*, chama a função não-recursiva, *string_tamanho*, com a string "Bíblia do Programador C/C++, do Jamsa!" 100.000 vezes, e, depois, exibe a quantidade de tempo necessária para realizar o processamento:

```
#include <stdio.h>
#include <time.h>

int string_tamanho(const char *str)
 {
   int tamanho = 0;

   while (*str++)
     tamanho++;
   return(tamanho);
 }

void main(void)
 {
   long int contador;
   time_t tempo_inicio, tempo_fim;
   time(&tempo_inicio);
   for (contador = 0; contador < 100000L; contador++)
     string_tamanho("Bíblia do Programador C/C++, do Jamsa!");
   time(&tempo_fim);
   printf("Tempo de processamento %d\n", tempo_fim - tempo_inicio);
 }
```

Em seguida, o programa *ok_recur.c*, usa uma implementação recursiva da função *string_tamanho* para realizar o mesmo processamento:

```
#include <stdio.h>
#include <time.h>

int string_tamanho(const char *str)
 {
   if (*str)
      return(1 + string_tamanho(str+1));
   else
      return(0);
 }

void main(void)
 {
```

```
  long int contador;

  time_t tempo_inicio, tempo_fim;
  time(&tempo_inicio);
  for (contador = 0; contador < 100000L; contador++)
    string_tamanho("Bíblia do Programador C/C++, do Jamsa!");
  time(&tempo_fim);
  printf("Tempo de processamento %d\n", tempo_fim - tempo_inicio);
}
```

Experimente esses programas, por exemplo, alterando o número de chamadas da função para um ou dois milhões. Como você descobrirá, a função não-recursiva é executada consideravelmente mais depressa que sua correspondente recursiva. Portanto, quando você projeta uma função recursiva, tenha em mente que pode acrescentar sobrecarga significativa ao tempo de execução do seu programa.

COMPREENDENDO POR QUE AS FUNÇÕES RECURSIVAS SÃO LENTAS 258

Uma função recursiva é uma função que chama a si mesma para realizar uma tarefa específica. Como você aprendeu na Dica 257, uma razão para evitar o uso da recursão é que as funções recursivas em geral são consideravelmente mais lentas que suas correspondentes não-recursivas. As funções recursivas são lentas porque a *sobrecarga da chamada* ocorre toda vez que a função é chamada. Como já foi detalhado na Dica 231, toda vez que seu programa chama uma função, o compilador C coloca na pilha o endereço do comando que segue imediatamente a chamada da função (chamado *endereço de retorno*). Em seguida, o compilador coloca os valores dos parâmetros na pilha. Quando a função termina, o sistema operacional do computador retira o endereço de retorno da pilha no contador de programa da CPU. Embora os computadores possam realizar essas operações muito rapidamente, elas ainda requerem tempo.

Como um exemplo, assuma que você chame a função *fatorial* recursiva com o valor 50. A função chamará a si mesma 49 vezes. Se cada função somar 10 milissegundos ao tempo de execução do seu programa, a função será meio segundo mais lenta que uma correspondente não-recursiva, que tem a sobrecarga de uma única chamada. Uma sobrecarga de meio segundo pode não parecer muito, porém, assuma que o programa chame a função 10 vezes. O retardo de meio segundo torna-se de cinco segundos. Se o programa usa a função 100 vezes, o retardo torna-se de 50 segundos, e assim por diante. Se você estiver desenvolvendo um programa que requer máximo desempenho, deverá tentar eliminar as funções recursivas sempre que possível.

Nota: Com os novos e mais rápidos microprocessadores (tais como os de 200 MHz), o retardo provocado pelas funções recursivas não é tão importante como era antigamente. No entanto, o impacto das funções recursivas ainda é significativo e você deve tentar escrever código eficiente e legível sem recursão sempre que possível.

COMPREENDENDO COMO REMOVER A RECURSÃO 259

Uma função recursiva é uma função que chama a si mesma para realizar uma tarefa específica. Como você aprendeu, é possível aumentar o desempenho do seu programa usando funções não-recursivas. Como regra, qualquer função que você pode escrever recursivamente, também pode escrever em termos de construções de laços de repetição, tais como um comando *for* ou *while*. O programa a seguir, *lacofato.c*, usa um laço *for* para implementar a função *fatorial*:

```
#include <stdio.h>

int fatorial(int valor)
 {
   int result = 1;
```

```
  int contador;

  for (contador = 2; contador <= valor; contador++)
    result *= contador;
  return(result);
}

void main(void)
{
  int i;

  for (i = 1; i <= 5; i++)
    printf("O fatorial de %d é %d\n", i, fatorial(i));
}
```

Sempre que você eliminar a recursão dentro de seus programas usando uma construção de laço de repetição, geralmente aumentará o desempenho do seu programa. No entanto, tenha em mente que os usuários podem compreender mais facilmente algumas operações que seus programas executarão quando você implementar as operações com recursão. Exatamente como algumas vezes você precisa escolher entre velocidade e consumo de memória do seu programa, algumas vezes precisa escolher entre legibilidade e desempenho.

260 PASSANDO STRINGS PARA AS FUNÇÕES

Como você aprendeu, quando passa parâmetros para as funções, C, por padrão, passa os parâmetros *por valor*. Portanto, quaisquer mudanças que sua função fizer nos parâmetros não existirão fora da função. Para alterar o valor de um parâmetro, você precisará passar o parâmetro *por referência*. A exceção a essa regra são as strings de caracteres. Ao chamar uma função com uma string de caracteres, você estará simplesmente passando uma matriz de byte para a função. Quando C passa uma matriz (qualquer tipo de matriz — não apenas uma string), C passa o endereço inicial da matriz para a função. Em outras palavras, C *sempre usa a chamada por referência para as matrizes*, de modo que você não precisa usar o operador de endereço.

261 PASSANDO ELEMENTOS ESPECÍFICOS DA MATRIZ

Na dica anterior você viu que C sempre passa as matrizes para as funções usando a chamada por referência. À medida que você trabalha com strings de caracteres, algumas vezes pode querer que uma função trabalhe com elementos específicos da matriz. Por exemplo, o programa a seguir, *meiamaiu.c* usa a função *strupr* para converter uma seção de uma string de caracteres para maiúsculas:

```
#include <stdio.h>
#include <string.h>

void main(void)
{
  char alfabeto[] = "abcdefghijklmnopqrstuvwxyz";

  strupr(&alfabeto[13]);
  printf(alfabeto);
}
```

A função *strupr* espera que o endereço inicial de uma string terminada por *NULL* seja um parâmetro. Neste caso, o programa passa para *strupr* o endereço da letra *n*, o qual a chamada da função então segue com vários caracteres terminados por *NULL*. Passando o endereço de um elemento de matriz específico, seus programas podem usar funções para manipular elementos específicos da matriz.

COMPREENDENDO CONST EM PARÂMETROS FORMAIS 262

Se você examinar os protótipos para as funções de manipulação de string anteriormente apresentados na seção Compreendendo as Strings, verá que muitas declarações de parâmetros colocam a palavra-chave *const* antes dos argumentos de string de caracteres, como mostrado aqui:

```
char *strcpy(char *destino, const char *origem);
```

No exemplo da definição da função *strcpy*, a palavra-chave *const* especifica que o código da função não deve mudar a variável *origem* dentro da função. Se o código de sua função tentar modificar o valor da string, o compilador gerará um erro. O programa a seguir, *chcconst.c*, usa a palavra-chave *const* para o parâmetro string:

```
#include <stdio.h>

void nao_muda(const char *string)
 {
   while (*string)
     *string++ = toupper(*string);
 }

void main(void)
 {
    char titulo[] = "Bíblia do Programador C/C++, do Jamsa!";

    nao_muda(titulo);
    printf(titulo);
 }
```

Como você pode ver, a função *nao_muda* tenta converter as letras da string para maiúsculas. No entanto, como o programa usa a palavra-chave *const*, o compilador exibirá uma mensagem de erro, e o código não compilará com sucesso. Você deverá usar a palavra-chave *const* diante de parâmetro que uma função recebe por referência quando não quiser que o valor do parâmetro seja modificado. Como C normalmente passa parâmetros que não são ponteiros por valor, os parâmetros por valor não requerem a palavra-chave *const*.

USAR CONST NÃO IMPEDE A MODIFICAÇÃO DO PARÂMETRO 263

Vimos na dica anterior que a palavra-chave *const* informa o compilador de que a função não deve modificar um valor de um parâmetro específico. Se uma função tentar modificar o valor de um desses parâmetros, o compilador gerará um erro, e o programa não será compilado. No entanto, você deve notar que só porque o cabeçalho da função especifica um parâmetro como uma constante, isso não significa que a função não pode modificar o valor do parâmetro. O programa a seguir, *mudacons.c*, usa um ponteiro para o parâmetro constante *string* para converter o conteúdo da string para maiúsculas:

```
#include <stdio.h>
#include <ctype.h>

void nao_muda(const char *string)
 {
   char *apelido = string;

   while (*apelido)
     *apelido++ = toupper(*apelido);
 }

void main(void)
 {
   char titulo[] = "Bíblia do Programador C/C++, do Jamsa!";

   nao_muda(titulo);
   printf(titulo);
 }
```

Quando você compilar e executar o programa *mudacons.c*, a função *nao_muda* converterá os caracteres da string para maiúsculas. Como você usou o apelido de ponteiros (referenciar as posições de memória de uma variável usando um nome diferente), o compilador não detecta a mudança no valor do parâmetro. Dependendo do tipo do seu compilador, ele poderá gerar uma mensagem de advertência. Se você estiver criando suas próprias funções, não use apelidos para alterar o valor de um parâmetro (como faz o programa *mudacons.c*.) Se um parâmetro é verdadeiramente uma constante, seu valor não deve mudar. O programa dentro desta dica deve ensinar você que a palavra-chave *const* não pode na verdade impedir que o valor de um parâmetro mude.

264 COMPREENDENDO AS DECLARAÇÕES DE STRINGS NÃO-LIMITADAS

Em C, uma string é uma matriz de valores de caracteres. Já vimos na seção Compreendendo as Strings que você especifica o número máximo de caracteres que a string conterá para criar uma string, como mostrado aqui:

```
char nome[64];
char titulo[32];
char buffer[512];
```

Quando você passa uma string de caracteres para uma função, na verdade passa o endereço inicial da string. Como o caractere *NULL* termina a string, as funções de C não se preocupam com quantos caracteres a string contém. Como resultado, muitas funções declaram parâmetros de strings de caractere como matrizes não-limitadas (matrizes que não especificam um tamanho), como mostrado aqui:

```
int strlen(char string[])
```

A declaração *char string[]* diz ao compilador que a função receberá um ponteiro para uma string terminada por *NULL*. A string pode conter 64 caracteres, 1.024 caracteres, ou talvez apenas o caractere *NULL*. O programa a seguir, *strmatriz.c*, usa uma matriz não-limitada para implementar a função *strlen*:

```
#include <stdio.h>

int strlen(char str[])
 {
   int i = 0;

   while (str[i] != NULL)
      i++;
   return(i);
 }

void main(void)
 {
   printf("O tamanho de ABC é %d\n", strlen("ABC"));
   printf("O tamanho de Bíblia do Programador C/C++ é %d\n",
     strlen("Bíblia do Programador C/C++"));
   printf("Tamanho de uma string NULL é %d\n", strlen(""));
 }
```

Quando você compilar e executar o programa *strmatriz.c*, verá que a função trabalhar para strings de qualquer tamanho. No entanto, como a maioria das funções que trabalha com strings, a função falhará se a string não for finalizada pelo caractere *NULL*.

265 USANDO PONTEIROS VERSUS DECLARAÇÕES DE STRING

À medida que você examinar diferentes funções que manipulam strings, verá strings de caracteres declaradas como matrizes não-limitadas ou como ponteiros, como mostrado aqui:

```
char *strcpy(char destino[], char origem[]);
char *strcpy(char *destino, char *origem);
```

Ambas as declarações no exemplo anterior informam o compilador de que ele está trabalhando com strings. Ambos são funcionalmente idênticos e ambos são corretos. Se você estiver criando suas próprias funções, o formato escolhido deverá depender de como você referencia o parâmetro dentro da função. Se você trata o parâmetro como um ponteiro, use a declaração no *estilo de ponteiro*. Se, em vez disso, você trata o parâmetro como uma matriz, use a matriz. Tratar o parâmetro de um modo consistente tornará seus programas mais fáceis de compreender.

COMO C USA A PILHA PARA OS PARÂMETROS STRING 266

Como você aprendeu, quando seus programas passam um parâmetro para as funções, C coloca o valor ou o endereço do parâmetro na pilha. Quando você passa uma string de caracteres para uma função, C coloca o endereço inicial da string na pilha. Por exemplo, a Dica 264 apresentou o programa *stmatriz.c*, que passou várias strings para a função *strlen*. A Figura 266 ilustra o valor do parâmetro que C coloca na pilha para a primeira chamada da função.

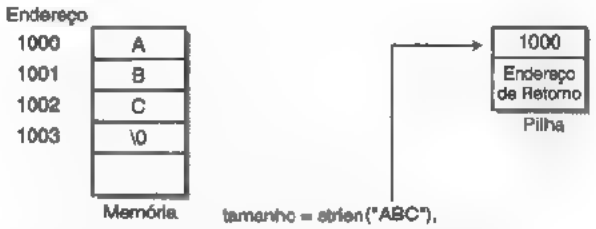

Figura 266 Como C passa parâmetros string para as funções.

Como você pode ver, C não coloca os caracteres da string na pilha. Em vez disso, C simplesmente coloca o endereço da string terminada por *NULL* na pilha. Como a função recebe somente um endereço (em vez de uma matriz de bytes), a função não se preocupa com quantos caracteres a string contém.

COMPREENDENDO AS VARIÁVEIS EXTERNAS 267

Frequentemente você pode usar as funções que cria para um programa dentro de outro. Para simplificar a reutilização das funções, os programadores geralmente colocam as funções em *bibliotecas de código-objeto*. A seção Ferramentas, mais à frente, discute o uso dessas bibliotecas. Em alguns casos, uma biblioteca pode definir uma variável global, tais como as variáveis *_fmode*, *_psp*, ou *errno* discutidas neste livro. Quando o código fora do programa atual define uma variável global e você quer usar a variável global dentro do seu programa, é preciso declarar a variável usando a palavra-chave *extern*. A palavra-chave *extern* diz ao compilador que outro programa declarou a variável *externamente* (fora do arquivo-fonte atual). Por exemplo, se você examinar o arquivo de cabeçalho *dos.h*, encontrará várias declarações de variáveis externas, incluindo as mostradas aqui:

```
extern int const _Cdecl _8087;
extern int _Cdecl _argc;
extern char **_Cdecl _argv;
extern char **_Cdecl ambiente;
```

Se você não usar a palavra-chave *extern*, o compilador assumirá que você está usando uma variável com o nome especificado. Por outro lado, quando você incluir a palavra-chave *extern*, o compilador procurará a variável global que você especificou.

COLOCANDO EXTERN EM USO 268

A Dica 267 introduziu a palavra-chave *extern*, que você usará dentro de seus programas para dizer ao compilador para usar uma variável global que outro programa declarou fora do programa atual. Para compreender melhor como a palavra-chave *extern* funciona, compile o arquivo *externo.c*, que contém a declaração da variável *conta_dica* e a função *exibe_titulo:*

```
#include <stdio.h>

int conta_dica = 1500;   // Variável global
```

```
void exibe_titulo(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

Quando você compilar e executar o programa *externo.c*, C criará o arquivo-objeto *externo.obj*. O programa *exibeext.c*, mostrado aqui, usa a variável externa *conta_dica* dentro do arquivo *externo.obj*:

```
#include <stdio.h>

void main(void)
 {
   extern int conta_dica;

   printf("O número de dicas é %d\n", conta_dica);
 }
```

Quando você compilar o programa *exibeext.c*, efetue os seguintes passos dentro do compilador *Turbo C++ Lite* (se não estiver usando o *Turbo C++ Lite*, confira a documentação do seu compilador):

1. Selecione a opção Open Project no menu Project.
2. Mude para o diretório que contém o programa *exibeext.c* e informe o nome do projeto como *exibeext*. Dê um clique com seu mouse em OK para criar o projeto.
3. Selecione a opção Add Item no menu Project.
4. Acrescente o arquivo *externo.obj* ao projeto.
5. Acrescente o arquivo *exibeext.c* ao projeto.
6. Selecione a opção Build All no menu Compile para criar o arquivo.

Neste caso, o programa *exibeext.c* exibe o valor da variável externa *conta_dica*. O programa não usa a função *exibe_titulo*, embora pudesse usar — bastaria chamar *exibe_titulo*. No entanto, o objetivo do programa foi ilustrar o uso da palavra-chave *extern*.

Nota: Para usar extern dentro de outro compilador, consulte a documentação on-line ou a impressa que veio com o compilador.

269 COMPREENDENDO A VARIÁVEL ESTÁTICA EXTERNA

A Dica 267 ensinou que a palavra-chave *extern* diz ao compilador C que você está referenciando uma variável global definida em um programa diferente em outro arquivo. Quando o linkeditor ligar os módulos do seu programa, ele determinará a posição de memória da variável. Na dica anterior você usou a variável global *conta_dica*, que foi definida no arquivo-objeto *externo.obj*. Como o programa *exibeext.c* referenciou a variável usando a palavra-chave *extern*, o programa poderia acessar a variável. Dependendo de seus programas, algumas vezes você usará variáveis globais em um arquivo-objeto que não quer funções fora do arquivo-objeto para acessar. Em tais casos, simplesmente preceda o nome da variável com a palavra-chave *static*:

```
static int nome_variável;
```

O arquivo a seguir, *estatext.c*, declara duas variáveis, uma chamada *dica_conta* e a outra *titulo*:

```
#include <stdio.h>

int conta_dica = 1500;  // Variável global
static char titulo[] = "Bíblia do Programador C/C++, do Jamsa!";

void exibe_titulo(void)
 {
   printf(titulo);
 }
```

Compile o arquivo *estatex.c* para criar o arquivo-objeto *estatex.obj*. Em seguida, crie o seguinte programa, *sem_esta.c*, que tenta usar ambas as variáveis contidas dentro do arquivo *estatex.obj*.

```
#include <stdio.h>

void main(void)
 {
   extern int conta_dica;
   extern char *titulo;
   void exibe_titulo(void);

   printf("O número de dicas é %d\n", conta_dica);
   printf("O título do livro é %s\n", titulo);
   exibe_titulo();
 }
```

Como você aprendeu na Dica 268, para compilar e vincular o programa usando o *Turbo C++ Lite*, siga o seguintes passos:

1. Selecione a opção Open Project no menu Project.
2. Mude para o diretório que contém o programa *sem_esta.c* e informe o nome do projeto como *sem_esta*. Dê um clique com seu mouse em OK para criar o projeto.
3. Selecione a opção Add Item no menu Project.
4. Acrescente o arquivo *estatex.obj* ao projeto.
5. Acrescente o arquivo *sem_esta.c* ao projeto.
6. Selecione a opção Build All no menu Compile para criar o arquivo.

Quando você compilar e ligar o programa *sem_esta.c*, o linkeditor deverá exibir uma mensagem dizendo que o compilador não pode resolver a variável *titulo*. Como a palavra-chave *static* precede a declaração da variável *titulo*, a variável somente é conhecida dentro do arquivo-objeto *extern2.obj*.

COMPREENDENDO A PALAVRA-CHAVE VOLATILE 270

À medida que a complexidade do seu programa aumenta, você pode eventualmente escrever funções e rotinas de baixo nível que acessam as portas de E/S do PC ou que servem os registradores de interrupção do PC (também referenciadas simplesmente como *interrupções*). Quando seus programas efetuam essas operações, usar uma interrupção ou acessar uma porta pode mudar suas variáveis que correspondem a posições de memória específicas ou endereços de portas. Como seu programa e muitos fatores externos ao seu programa podem mudar muitas variáveis, você precisa dizer ao compilador de que o valor da variável pode mudar a qualquer momento. Para informar o compilador que as operações fora do programa podem alterar o valor de uma variável, use a palavra-chave *volatile*, como mostrado aqui:

```
volatile int alguma_variavel;
```

Quando o compilador encontrar a palavra-chave *volatile*, ele saberá que não pode fazer suposições sobre o valor da variável a qualquer tempo. Por exemplo, o compilador não colocará o valor da variável em um registrador para acesso rápido. Fazer isso incorreria no risco de o valor do registrador não ser o mesmo que o conteúdo da memória da variável, que uma interrupção (por exemplo) poderia ter alterado após o armazenamento da variável no registrador, sem o conhecimento do programa. Em vez disso, quando o programa precisar acessar o valor de uma variável, ele especificamente referenciará a posição de memória da variável.

Nota: Você geralmente deve declarar as variáveis volatile como variáveis globais. Deste modo, os programas e as operações de fora referenciam posições de memória contidas dentro do segmento de dados do programa, e não as posições na pilha, que o programa descarta quando a função correspondente termina.

COMPREENDENDO A ESTRUTURA DE CHAMADA E O PONTEIRO DE BASE 271

Você aprendeu que, quando seu programa chama uma função, C coloca o endereço de retorno e os parâmetros da função na pilha. Dentro da pilha, C referencia as informações salvas da função chamada como *quadro da chamada* (*call frame*). Para ajudar suas funções a localizar rapidamente o quadro da chamada, C atribui o registrador

Ponteiro Base (BP) ao endereço do início do quadro. C também coloca as variáveis locais da função na pilha (dentro do quadro da pilha). A Figura 271 ilustra o conteúdo de um único quadro da pilha:

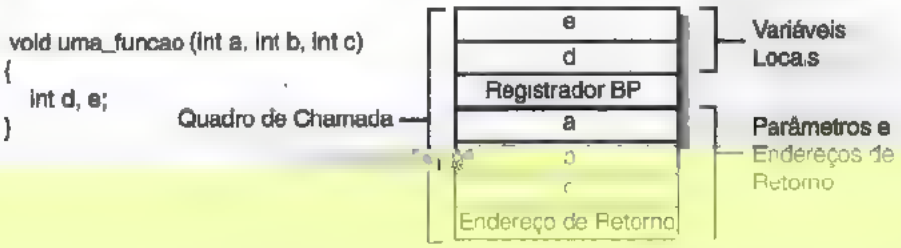

Figura 271 As informações que C coloca na pilha para uma chamada de função constituem um quadro da chamada

Quando você escreve funções em linguagem assembly que serão chamadas dentro de seus programas em C, precisa compreender o uso e a estrutura dos quadros da pilha para que suas funções em Assembly possam acessar os valores dos parâmetros dentro desse quadro da pilha.

272 CHAMANDO UMA FUNÇÃO EM LINGUAGEM ASSEMBLY

Na Dica 230 você viu que seus programas podem chamar funções escritas em outras linguagens de programação, tais como Pascal. Além disso, seus programas podem chamar rotinas em linguagem assembly. A rotina em linguagem assembly a seguir, *troca_valores*, permita os valores de duas variáveis passadas para a função por referência (por endereço):

```
                MODEL    small
               .CODE
               PUBLIC     troca_valores

  troca_valores     PROC
               push bp
               mov  bp,sp
               sub  sp,2
               push si
               push di
               mov  si,word ptr [bp+4]        ;Arg1
               mov  di,word ptr [bp+6]        ;Arg2

               mov  ax,word ptr [si]
               mov  word ptr [bp-2],ax

               mov  ax,word ptr[di]
               mov  word ptr [si],ax

               mov  ax,word ptr [bp-2]
               mov  word ptr [di],ax

               pop  di
               pop  si
               mov  sp,bp
               pop  bp
 saida:        ret
  troca_valores     ENDP
               END
```

O CD-ROM que acompanha este livro contém o arquivo *troca.asm*. Se você estiver usando o Borland C++, monte o arquivo para criar o arquivo objeto *troca.obj* como mostrado aqui:

```
C:\> TASM TROCA.ASM  <Enter>
```

Em seguida, crie o seguinte programa C, *usa_troca.c*, que usa a função *troca_valores*:

```
#include <stdio.h>

void troca_valores(int *, int *);

void main(void)
 {
   int a = 1, b = 2;

   printf('Valores originais a %d  b %d\n", a, b);
   troca_valores(&a, &b);
   printf('Valores permutados a %d  b %d\n", a, b);
 }
```

Neste caso, você escreveu a função *troca_valores* para suportar ponteiros do tipo *near*. Se você alterar os modelos de memória, precisará alterar a rotina em linguagem assembly.

RETORNANDO UM VALOR A PARTIR DE UMA FUNÇÃO EM LINGUAGEM ASSEMBLY 273

Na Dica 261 você aprendeu como chamar uma função em linguagem assembly a partir de dentro de um programa C. No programa *troca.asm*, a função não retornou um resultado. No entanto, a seguinte rotina em linguagem assembly, *pega_maximo*, retorna o maior dentre dois valores inteiros:

```
                .MODEL small
                .CODE
                PUBLIC _pega_maximo
_pega_maximo            PROC
                push bp
                mov  bp,sp

Arg1                 equ  [bp+4]
Arg2                 equ  [bp+6]

                mov  ax,Arg1          ;Move Arg1 para AX
                cmp  Arg2,ax          ;Compara Arg2 com Arg1
                jg   arg2_maior       ;Desvia se Arg2 for maior
                jmp  terminado

arg2_maior:     mov  ax,Arg2

terminado:      pop  bp
                ret
_pega_maximo         ENDP
                END
```

O CD-ROM que acompanha este livro contém o arquivo *pega_max.asm*, que contém a rotina *pega_maximo*. Como você pode ver, a rotina em linguagem assembly coloca seu resultado no registrador AX. Dicas posteriores explicam os diferentes registradores em detalhe; por enquanto, você pode considerar o registrador AX como similar ao registrador BP, que a dica anterior explicou. O seguinte programa C, *usa_max.c*, chama a função da linguagem assembly para determinar o maior dentre dois valores:

```
#include <stdio.h>
extern int pega_maximo(int, int);

void main(void)
 {
   int result;

   result = pega_maximo(100, 200);
   printf('O maior valor é %d\n', result);
 }
```

Quando o programa chamar a função, o compilador C atribuirá o valor do registrador AX como o resultado da função.

274 INTRODUZINDO FUNÇÕES QUE NÃO RETORNAM VALORES

À medida que o número de funções que você cria for aumentando, eventualmente você criará uma função que não retornará um valor. Como visto, o compilador C, a não ser que instruído de outra forma, assume que uma função retorna o tipo *int*. Se suas funções não retornam um valor, você deve declarar a função como tipo *void*, como mostrado aqui:

```
void minha_func(int idade, char *nome);
```

Se o programa mais tarde tentar usar o valor de retorno da função, como mostrado aqui, o compilador gerará um erro:

```
result = minha_func(32, "Jamsa");
```

275 COMPREENDENDO AS FUNÇÕES QUE NÃO USAM PARÂMETROS

Aumentando o número de programas e funções criado por você, é possível, eventualmente, criar uma função que não usa nenhum parâmetro. Ao definir a função (e o protótipo da função), você deverá usar a palavra-chave *void* para informar o compilador (e outros programadores) de que a função não usa parâmetros:

```
int minha_func(void);
```

Se o programa mais tarde chamar a função com parâmetros, o compilador gerará um erro.

276 COMPREENDENDO A PALAVRA-CHAVE AUTO

À medida que você for examinando programas C, encontrará declarações de variáveis que usam a palavra-chave *auto*, como mostrado aqui:

```
auto int contador;
auto int sinaliz;
```

A palavra-chave *auto* informa o compilador de que a variável é local na função, e que o compilador deve automaticamente criar e destruir a variável. O compilador cria variáveis automáticas alocando espaço na pilha. Como as variáveis são automáticas por padrão, a maioria dos compiladores omite a palavra-chave *auto*. Dentro de uma função, as seguintes declarações de variáveis são idênticas:

```
auto int contador;
int contador;
```

277 COMPREENDENDO O ESCOPO

Dentro de seu programa, as funções e as variáveis têm um escopo que define as áreas dentro do programa onde seus nomes têm significado. Por exemplo, considere o seguinte programa, *doiscont.c*, que usa duas variáveis chamadas *conta*.

```
#include <stdio.h>

void alarme(int conta_bip)
 {
   int conta;

   for (conta = 1; conta <= conta_bip; conta++)
     putchar(7);
 }
```

```
void main(void)
 {
   int conta;
   for (conta = 1; conta <= 3; conta++)
    {
       printf("Prestes a soar %d vezes \n", conta);
       alarme(conta);
    }
 }
```

Como você pode ver, ambas as funções *alarme* e *main* usam variáveis chamadas *conta*. No entanto, para C, ambas as variáveis são distintas — cada uma tem um escopo diferente. No caso da função *alarme*, C somente conhece sua variável *conta* (isto é, *conta* tem um escopo definido) enquanto a função estiver em execução. Da mesma forma, no caso de *main*, sua variável *conta* somente tem significado enquanto *main* estiver em execução. Como resultado, o laço *for* que muda a variável *conta* na função *alarme* não tem efeito na variável *conta* dentro de *main*.

Quando você discute o escopo de uma variável, freqüentemente usa os termos variáveis *locais* e *globais*. Uma variável local tem seu escopo restrito a uma função específica. Por outro lado, o programa inteiro conhece uma variável global. No caso do programa *doiscont.c*, cada função define cada ocorrência da variável *conta* como local.

COMPREENDENDO AS CATEGORIAS DE ESCOPO DE C 278

Como você aprendeu, o *escopo* de um identificador (geralmente o nome de uma variável ou função) é a parte do programa dentro do qual o identificador tem significado (em outras palavras, onde o programa pode usar o identificador). C define quatro categorias de escopo: bloco, função, protótipo de função e arquivo. Adicionalmente, C++ define o escopo de classe. O *escopo de bloco* define a região delimitada por chaves dentro da qual seu programa definiu uma variável. Normalmente, o escopo de bloco referencia uma função. As variáveis locais têm escopo no bloco. No entanto, como já vimos na seção "Introdução à Linguagem C", você pode declarar variáveis após qualquer abre chaves. O escopo da variável existe até o fecha chaves — o que significa que um parâmetro com o escopo de bloco somente pode ter escopo dentro de um *if* condicional. Os parâmetros formais têm escopo de bloco, com o escopo limitado à função que define o parâmetro. O *escopo da função* define a região entre o abre e fecha chaves de uma função. O único item com escopo de função é um rótulo usado pelo comando *goto*. O *escopo de protótipo de função* especifica a região dentro do início e fim de um protótipo de função. Os identificadores que aparecem dentro de um protótipo de função têm significado somente dentro do protótipo da função, como mostrado aqui:

```
int uma_funcao(int idade, char *nome);
```

O *escopo de arquivo* especifica uma região da declaração do identificador até o final do arquivo-fonte. As variáveis globais têm escopo de arquivo, o que significa que somente funções que seguem fisicamente a declaração da variável global dentro do arquivo podem referenciar uma variável global. Em C++, o *escopo de classe* define a coleção nomeada de métodos e a estrutura de dados que compõem a classe.

COMPREENDENDO O ESPAÇO DO NOME E OS IDENTIFICADORES 279

Como você sabe, o *escopo* define a região de um programa dentro do qual um identificador tem significado. Similarmente, o espaço do nome define uma região dentro da qual os nomes de identificadores precisam ser exclusivos. No sentido mais simples, um *identificador* é um nome. C define quatro classes de identificadores, como mostrado na lista a seguir:

- Nomes de rótulos em *goto:* os nomes de rótulo usados em um comando *goto* precisam ser únicos dentro de uma função.
- Estrutura, união e tags de enumeração: um *tag* é um nome de uma estrutura, união ou tipo enumerado. Os tags precisam ser únicos dentro de um bloco.

- Nomes de membro de estrutura e de união: os nomes de membro que aparecem dentro de uma estrutura ou união precisam ser únicos. Diferentes uniões ou estruturas podem ter os mesmos nomes de membro.
- Variáveis, identificadores *typedef*, funções e membros enumerados: esses identificadores precisam ser únicos dentro do escopo (como explicado na Dica 278) no qual o identificador está definido.

280 COMPREENDENDO A VISIBILIDADE DO IDENTIFICADOR

Como você aprendeu, o *escopo* define a região do programa dentro da qual um identificador tem significado. De um modo similar, a *visibilidade* de um identificador define a região do código dentro do qual um programa pode acessar um identificador. Normalmente, o escopo e a visibilidade de um identificador são os mesmos. No entanto, quando seu programa declara um identificador com o mesmo nome dentro de um bloco que aparece dentro do escopo de um identificador existente, o compilador temporariamente esconde o identificador externo (em outras palavras, o identificador externo perde a visibilidade e o compilador não o reconhece). Considere o seguinte programa, *visivel.c*, que usa dois identificadores chamados *valor*:

```
#include <stdio.h>

void main(void)
 {
   int valor = 1500;
   if (valor > 1499)
     {
       int valor = 1;

       printf("O valor interno é %d\n", valor);
     }
   printf("O valor externo é %d\n", valor);
 }
```

Quando você compilar e executar o programa *visivel.c*, sua tela exibirá o seguinte:

```
O valor interno é 1
O valor externo é 1500
C:\>
```

Quando o programa declara a variável *valor* dentro do comando *if*, a declaração da variável instrui o compilador a esconder a ocorrência externa da variável com o mesmo nome. No entanto, fora do bloco, a variável externa torna-se novamente visível para o compilador.

281 COMPREENDENDO A DURAÇÃO

Quando você discute variáveis, *duração* especifica a quantidade de tempo durante a qual um identificador possui memória alocada pelo sistema. C suporta três tipos de duração: *local*, *estática* e *dinâmica*. As variáveis estáticas criadas durante a chamada de uma função ou variáveis definidas dentro de um bloco de comandos têm *duração local*. Seus programas sempre precisam inicializar variáveis locais. Caso seu programa não inicialize uma variável local, ele não pode prever o conteúdo da variável. O compilador cria *variáveis estáticas* à medida que a execução do programa inicia. As variáveis estáticas normalmente correspondem às variáveis globais. A maioria dos compiladores C inicializa as variáveis estáticas com 0. O compilador aloca *variáveis dinâmicas* a partir do heap durante a execução do programa. Na maioria dos casos, os programas precisam inicializar as variáveis dinâmicas.

Nota: Algumas funções da biblioteca de execução inicializarão as posições de memória com 0 (zero), enquanto outras não.

FUNÇÕES QUE SUPORTAM UM NÚMERO VARIÁVEL DE PARÂMETROS 282

Já foi visto que C mapeia os parâmetros reais que passa para uma função com os parâmetros formais definidos no cabeçalho da função. Se a função espera três parâmetros, a chamada de sua função deve incluir três valores de parâmetro. Caso considere funções tais como *printf* ou *scanf*, no entanto, você verá que as funções suportam um número variável de parâmetros. Por exemplo, as seguintes chamadas à função *printf* são todas válidas:

```
printf("Bíblia do Programador C/C++, do Jamsa!");
printf("%d %d %d %d %d", 1, 2, 3, 4, 5);
printf("%f %s %s %d %x", salario, nome, estado, idade, id);
```

Como você verá a seguir, na Dica 283, você pode usar as macros *va_arg*, *va_end* e *va_start* (definidas no arquivo de cabeçalho *stdarg.h*) para instruir seus programas a criar suas próprias funções que suportam um número variável de parâmetros. As macros essencialmente retiram parâmetros da pilha, um de cada vez, até que o programa chegue ao último parâmetro. Ao usar essas macros para obter os parâmetro, você precisará conhecer o tipo de cada parâmetro. No caso de *printf*, a função usa os especificadores de formato (por exemplo, *%d*, *%s* e *%f*) para controlar os tipos de parâmetros.

SUPORTANDO UM NÚMERO VARIÁVEL DE PARÂMETROS 283

Nesta dica, você criará uma função chamada *soma_valores*, que soma todos os valores inteiros que a função chamadora passa para ela. Como mostrado aqui, a função suporta um número variável de parâmetros. O valor 0 dentro da chamada da função indica o último parâmetro (o que não afeta a soma):

```
result = soma_valores(3,0);                    // Retorna 3
result = soma_valores(3,5,0);                  // Retorna 8
result = soma_valores(100,3,4,2,0);            // Retorna 109
```

O programa a seguir, *advalor.c*, contém e usa a função *soma_valores*:

```
#include <stdio.h>
#include <stdarg.h>

int soma_valores(int valor, ...)
 {
   va_list argumento_ptr;
   int result = 0;

   if (valor != 0)
     {
       result += valor;
       va_start(argumento_ptr, valor);
       while ((valor = va_arg(argumento_ptr, int)) != 0)
         result += valor;
       va_end(argumento_ptr);
     }
   return(result);
 }

void main(void)
 {
   printf("A soma de 3 é %d\n", soma_valores(3, 0));
   printf("A soma de 3 + 5 é %d\n", soma_valores(3, 5, 0));
   printf("A soma de 3 + 5 + 8 é %d\n", soma_valores(3, 5, 8, 0));
   printf("A soma de 3 + 5 + 8 + 9 é %d\n", soma_valores(3, 5, 8, 9, 0));
 }
```

A função *soma_valores* usa a macro *va_start* para atribuir um ponteiro (*argumento_ptr*) ao primeiro parâmetro na pilha. Em seguida, a função usa a macro *va_arg* para obter os valores um de cada vez. A macro *va_arg*

retorna um valor do tipo especificado, e, depois, incrementa o *argumento_ptr* para apontar para o próximo argumento. Quando o *argumento_ptr* encontra o zero finalizador, a função usa a macro *va_end* para atribuir um valor ao *argumento_ptr* que impede o uso futuro de *argumento_ptr* (até que *va_start* reinicialize o *argumento_ptr*). Quando você criar funções que suportem um número variável de parâmetros, suas funções precisarão ter um modo de conhecer o número de parâmetros e o tipo de cada um. No caso de *printf*, o especificador de formato define os parâmetros e seus tipos. No caso de *soma_valores*, o finalizador zero marca o último parâmetro. Da mesma forma, todos os argumentos passados para a função são do mesmo tipo.

Nota: Observe o uso das reticências (. .) dentro do cabeçalho da função soma_valores para indicar um número variável de parâmetros.

284 COMO VA_START, VA_ARG E VA_END FUNCIONAM

Na dica anterior você aprendeu que pode usar as macros *va_start*, *va_arg* e *va_end*, definidas dentro do arquivo de cabeçalho *stdarg.h*, para criar funções que suportam um número variável de parâmetros. Para compreender melhor como essas macros funcionam, considere a seguinte chamada à função *soma_valores:*

```
soma_valores(10, 20, 30, 0);
```

Quando o programa fizer a chamada da função, o compilador colocará os parâmetros na pilha da direita para a esquerda. Dentro da função, a macro *va_start* atribui um ponteiro para o primeiro parâmetro, como mostrado na Figura 284.

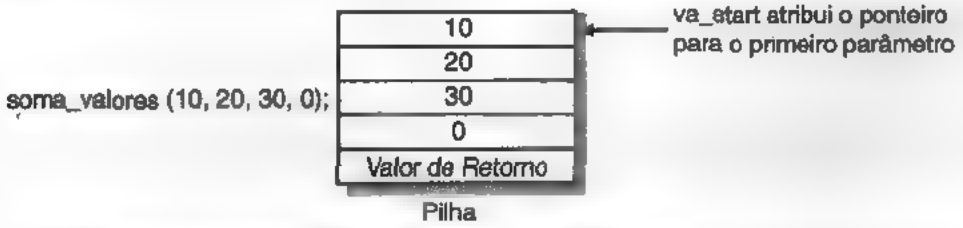

Figura 284 Usando *va_start* para atribuir um ponteiro ao primeiro parâmetro.

A macro *va_arg* retorna o valor apontado pelo argumento ponteiro. Para determinar o valor, a macro precisa, conhecer o tipo do parâmetro. Um parâmetro do tipo *int*, por exemplo, usará 16 bits, enquanto um parâmetro do tipo *long* usará 32 bits. Após recuperar o valor do parâmetro, a macro *va_arg* incrementará o argumento ponteiro para que ele aponte para o próximo argumento. Para determinar o número de bytes a adicionar ao ponteiro, *va_arg* usará novamente o tipo do parâmetro. Após a macro *va_arg* recuperar o último argumento, a macro *va_end* nulificará o valor do argumento ponteiro.

285 CRIANDO FUNÇÕES QUE SUPORTAM MUITOS PARÂMETROS E TIPOS

Nas Dicas 282 e 283 você aprendeu como criar funções que suportam um número variável de parâmetros. Infelizmente, a função *soma_valores* que você criou suporta somente parâmetros do tipo *int*. O programa a seguir, *todotipo.c*, modifica a função *soma_valores* para suportar valores de todos os tipos. A função retorna um valor do tipo *float*. Para ajudar a função a determinar os tipos dos parâmetros, você passa para a função um especificador de formato similar ao usado por *printf*. Por exemplo, para somar três valores *inteiros*, use a seguinte chamada:

```
result = soma_valores("%d %d %d", 1, 2, 3);
```

Da mesma forma, para somar três valores diferentes em ponto flutuante, use a seguinte chamada:

```
result = soma_valores("%f %f %f", 1.1, 2.2, 3.3);
```

Finalmente, para somar valores *inteiros* e *pontos-flutuantes*, use a seguinte chamada:

```
result = soma_valores("%f %d %f %d", 1.1, 2, 3.3, 4);
```

Usando o especificador de formato, você elimina a necessidade do zero finalizador. Adicionalmente, o especificador de formato lhe permite determinar quantos bits cada parâmetro usa, como mostrado aqui:

```
#include <stdio.h>
#include <stdarg.h>

double soma_valores(char *str, ...)
 {
   va_list marcador;
   double result = 0.0;

   va_start(marcador, str); // marca o primeiro argumento adicional
   while (*str)             // examina cada caractere na string
    {
     if (*str == '%')       // se não for um especificador de formato %-, pula
       {
         switch (*(++str))
           {
           case 'd': result += va_arg(marcador, int);
                 break;
           case 'f': result += va_arg(marcador, double);
                 break;
           }
       }
     str++;
    }
   va_end(marcador);
   return(result);
 }

void main(void)
 {
   double result;

   printf("Result %f\n", soma_valores("%f", 3.3));
   printf("Result %f\n", soma_valores("%f %f",1.1,2.2));
   printf("Result %f\n", soma_valores("%f %d %f",1.1, 1, 2.2));
   printf("Result %f\n", soma_valores("%f %d %f %d", 1.1, 1, 2.2, 3));
 }
```

LENDO UM CARACTERE DO TECLADO 286

Até mesmo os mais simples programas precisam freqüentemente ler caracteres do teclado. O caractere pode corresponder a uma opção de menu, uma resposta Sim ou Não, ou até uma das muitas letras em um nome. Os programas normalmente efetuam as operações de entrada de caractere usando a macro *getchar*. Você implementará a macro *getchar* como mostrado aqui:

```
#include <stdio.h>

int getchar(void);
```

Se for bem-sucedida, *getchar* retorna o valor ASCII para o caractere lido. Se um erro ocorrer ou *getchar* encontrar um final de arquivo (normalmente para entrada redirecionada), *getchar* retorna *EOF*. O programa a seguir, *getchar.c*, usa *getchar* para ler uma resposta Sim ou Não do teclado:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
 {
```

```
  int letra;

  printf("Digite S ou N para continuar e pressione Enter\n");
  do
   {
    letra = toupper(getchar());
   }
  while ((letra != 'S') && (letra != 'N'));
  printf("Você digitou %c\n",((letra == 'S') ? 'S': 'N'));
 }
```

Como você pode ver, o programa usa um laço *do while* para repetidamente chamar *getchar* até que o usuário digite S ou N no teclado.

Nota: Para suportar o redirecionamento da E/S, C realmente define a macro **getchar** *em termos de* **stdin** *(o que corresponde ao teclado por padrão).*

287 EXIBINDO UM CARACTERE DE SAÍDA

Na dica anterior você aprendeu como usar a macro *getchar* para ler um caractere do teclado. De um modo similar, C fornece a macro *putchar*, que escreve um caractere na tela (*stdout*). O formato da macro *putchar* é:

```
#include <stdio.h>

int putchar (int letra);
```

Se *putchar* for bem-sucedida, ela retornará o caractere escrito. Se ocorrer um erro, *putchar* retorna *EOF*. O programa a seguir, *putchar.c*, usa *putchar* para exibir as letras do alfabeto:

```
#include <stdio.h>

void main(void)
 {
   int letra;

   for (letra = 'A'; letra <= 'Z'; letra++)
      putchar(letra);
 }
```

Nota: Como C define **putchar** *em termos de* **stdout**, *você pode usar os operadores de redirecionamento do DOS para redirecionar a saída do programa* **putchar** *para um arquivo ou para a impressora.*

288 COMPREENDENDO A ENTRADA BUFFERIZADA

Quando seu programa usa entrada bufferizada, o sistema operacional não passa as letras que o usuário digita para o programa até que ele pressione Enter. Deste modo, o usuário pode modificar os caracteres à medida que digitar, usando a tecla Backspace (Retrocesso) para apagar os caracteres conforme necessário. Quando o usuário pressiona Enter, todos os caracteres digitados estão disponíveis para o programa. A macro *getchar* usa entrada bufferizada. Se você usar *getchar* para ler uma resposta de um único caractere, *getchar* não irá ler um caractere até que o usuário pressione Enter. Se o usuário digitar múltiplos caracteres, todos os caracteres estarão disponíveis para *getchar* dentro do buffer de entrada. O programa a seguir, *es_buff.c*, ilustra a entrada bufferizada. Execute o programa, e, depois, digite uma linha de texto. Os caracteres que você digitar não estarão disponíveis para o programa até que você pressione Enter. No entanto, após você pressionar Enter, o programa irá ler e exibir caracteres até encontrar o caractere de nova linha (que o sistema operacional cria quando você pressiona Enter), como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   int letra;
```

```
    do
     {
      letra = getchar();
      putchar(letra);
     {
    while (letra != '\n');
}
```

Quando você executar o programa *es_buff.c*, experimente com as letras que você digitar, usando a tecla Backspace para apagar letras. Como verá, as letras passadas para o programa correspondem ao seu texto final.

ATRIBUINDO ENTRADA DO TECLADO A UMA STRING 289

A seção Compreendendo as Strings deste livro discute vários modos diferentes de manipular strings. Quando você efetuar entrada no teclado, uma das operações mais comuns que seus programas fará é atribuir os caracteres que resultam da entrada no teclado a uma string. O programa a seguir, *preenche.c*, usa a macro *getchar* para atribuir letras à variável *string*. Para atribuir os caracteres, o programa simplesmente fica em um laço, atribuindo caracteres aos elementos da string até encontrar o caractere de *nova linha*. O programa então atribui o caractere marcador *NULL* (final da string) à posição atual da string, como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   char string[128];
   int indice = 0;
   int letra;

   printf("Digite uma string e pressione Enter \n");
   while ((letra = getchar()) != '\n')
      string[indice++] = letra;
   string[indice] = NULL;
   printf("A string foi: %s\n", string);
 }
```

COMBINANDO GETCHAR E PUTCHAR 290

Como você aprendeu, *getchar* lhe permite ler do teclado (*stdin*), enquanto *putchar* lhe permite exibir uma letra na tela (*stdout*). Dependendo da função do seu programa, algumas vezes você poderá querer ler e exibir caracteres. Por exemplo, o laço *do while*, a seguir, irá ler e exibir caracteres até e incluindo o caractere de *nova linha*:

```
do
   {
     letra = getchar();
     putchar(letra);
   }
while (letra != '\n');
```

Como *getchar* e *putchar* trabalham com valores inteiros, você pode combinar os comandos anteriores, como mostrado aqui:

```
do
   putchar(letra = getchar());
while (letra != '\n');
```

Neste caso, *getchar* atribuirá o caractere digitado à variável *letra*. A macro *putchar*, por sua vez, exibirá o valor atribuído a *letra*.

291 LEMBRE-SE, GETCHAR E PUTCHAR SÃO MACROS

À medida que você cria seus programas, lembre-se de que *getchar* e *putchar* são macros, não funções. Portanto, alguns compiladores não lhe permitem deixar espaços entre os nomes dessas macros e os parênteses, como mostrado aqui:

```
letra = getchar()

putchar(letra);
```

Se você examinar o arquivo de cabeçalho *stdio.h*, encontrará as definições de macros para *getchar* e *putchar*. A seção Redirecionando a E/S e Processando Linhas de Comando, mais à frente, explica as definições das macros *getchar* e *putchar* em detalhes.

292 LENDO UM CARACTERE USANDO E/S DIRETA

Conforme você aprendeu na Dica 288, quando seus programas recebem entrada no teclado, eles podem usar a entrada direta ou bufferizada. Quando seus programas usam as operações de entrada direta, os caracteres que o usuário digita no teclado ficam imediatamente disponíveis para o programa (em outras palavras, o sistema operacional não bufferiza os caracteres). Se o usuário pressionar a tecla Backspace para apagar um caractere anterior, o próprio programa precisará tratar a operação de edição (apagando o caractere anterior da tela e removendo o caractere do buffer). A função *getche* permite que seus programas leiam um caractere do teclado usando entrada direta. O formato da função *getche* é:

```
#include <conio.h>
int getche(void)
```

O programa a seguir, *getche.c*, usa a função *getche* para ler uma resposta Sim ou Não do teclado:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

void main(void)
 {
   int letra;

   printf("Quer continuar? (S/N): ");
   do
     {
       letra = getche();
       letra = toupper(letra);
     }
   while ((letra != 'S') && (letra != 'N'));

   if (letra == 'S')
      printf("\nSua resposta foi Sim\n");
   else
      printf("\nPor que não?\n");
 }
```

Ao contrário do programa *getchar.c*, que requer que o usuário pressione Enter para tornar a resposta disponível, as teclas que o usuário digitou no programa *getche.c* ficam imediatamente disponíveis para o programa.

293 ENTRADA DIRETA NO TECLADO SEM A EXIBIÇÃO DO CARACTERE

Vimos na dica anterior como usar a função *getche* para ler caracteres do teclado à medida que ele digita os caracteres (usando a E/S direta). Quando você usar *getche*, o programa exibirá as letras que o usuário digita na tela à medida que ele as digitar. Dependendo do seu programa, algumas vezes você poderá querer ler caracteres do teclado sem exibir os caracteres na tela. Por exemplo, se seu programa pedir que o usuário informe uma senha, as letras que ele digitar não deverão aparecer na tela para que os outros vejam. A função *getch* permite que seus programas leiam caracteres do teclado sem exibir (ecoar) os caracteres na tela. O formato da função *getch* é este:

```
#include <conio.h>

int getch(void);
```

O programa a seguir, *getch.c*, usa a função *getch* para ler caracteres do teclado. À medida que o usuário digita, o programa usa *getch* para ler cada caractere, converte cada caractere para maiúsculo, e, depois, exibe o equivalente maiúsculo de cada caractere na tela. O programa a seguir, *getch.c*, mostra como você pode rapidamente implementar esse processamento:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main(void)
 {
   int letra;

   printf("Digite uma string de caracteres e pressione Enter\n");
   do
   {
     letra = getch();
     letra = toupper(letra);
     putch(letra);
   }
    while (letra != '\r');
 }
```

SABENDO QUANDO USAR '\R' E '\N' 294

Como você aprendeu, C usa a seqüência de escape '\r' para indicar um avanço do carro. Da mesma forma, C usa '\n' para representar uma *nova linha* (retorno do carro e alimentação de linha). Quando seus programas executam entrada bufferizada usando *getchar*, C converte a tecla Enter para a sequência de retorno do carro e alimentação de linha. Por outro lado, quando você executa E/S direta usando *getch* ou *getche*, ambas as funções retornam a tecla Enter simplesmente como um retorno do carro ('\r'). Portanto, você precisa testar o caractere correto dentro de seus programas, como mostrado aqui:

```
do
   {
     letra = getchar();
     putchar(letra);
   }
while (letra != '\n');
do
   {
     letra = getch();
     putchar(letra);
   }
while (letra != '\r');
```

EXECUTANDO SAÍDA DIRETA 295

Já vimos que, as funções *getch* e *getche* permitem que seus programas leiam caracteres diretamente do teclado, ignorando os canais de entrada bufferizada de C (baseados no sistema de arquivos). De uma forma similar, seus programas podem efetuar saída rápida na tela usando a função *putch*, como mostrado aqui:

```
#include <conio.h>

int putch(int letra);
```

Se bem-sucedida, *putch* retorna a letra que exibiu. Se um erro ocorrer, *putch* retorna *EOF*. Para efetuar saída rápida, a função *putch* comunica-se com os serviços de vídeo da BIOS ou acessa diretamente a memória de

vídeo do PC. Funções tais como *putchar*, por outro lado, usam o sistema de arquivos, que por sua vez chama a BIOS. A função *putch* não converte um caractere de alimentação de linha em uma seqüência de retorno do carro e alimentação de linha. O programa a seguir, *putch.c*, usa *putch* e *putchar* para exibir as letras do alfabeto 1.001 vezes. O programa então exibe a quantidade de tempo que cada função requer, como mostrado aqui:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
 {
   int letra;
   int conta;

   time_t hora_inicio, hora_parada;
   time(&hora_inicio);
   for (conta = 0; conta < 1000; conta++)
     for (letra = 'A'; letra <= 'Z'; letra++)
       putchar(letra);
   time(&hora_parada);
   printf("\n\nTempo requerido por putchar %d segundos\n", hora_parada
   -hora_inicio);
   printf("Pressione qualquer tecla...\n");
   getch();
   time(&hora_inicio);
   for (conta = 0; conta < 1000; conta++)
      for (letra = 'A'; letra <= 'Z'; letra++)
        putch(letra);
   time(&hora_parada);
   printf("\n\nTempo requerido por putch %d segundos\n", hora_parada
   -hora_inicio);
 }
```

296 COLANDO UMA TECLA DIGITADA DE VOLTA NO BUFFER DO TECLADO

Como você aprendeu, a função *getch* permite que seus programas leiam um caractere do teclado. Dependendo de como você escreve seus programas, algumas vezes você lê caracteres até um caractere específico, e, depois, processa as teclas digitadas. Quando o processamento termina, você lê os caracteres restantes. Quando você escrever esse código, algumas vezes poderá querer que seu programa "anule a leitura" de um caractere. A função *ungetch* permite que seus programas "anulem a leitura" de um caractere. Para fazer isso, você implementará a função *ungetch*, como mostrado aqui:

```
#include <conio.h>

int ungetch(int caractere);
```

Além disso, algumas vezes você pode querer colocar um caractere no buffer do teclado para que seu programa possa reler a tecla digitada que acabou de ler. Usando *ungetch*, seus programas podem fazer exatamente isso. O programa a seguir, *ungetch.c*, lê letras do teclado até que encontre uma letra não-minúscula. O programa então exibe as letras, e, depois disso, lê e exibe quaisquer caracteres restantes em uma linha diferente:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
```

```
void main(void)
 {
   int letra;
   int feito = 0;
   int maiusc_encontrado = 0;

   do
    {
     letra = getch();
     if (islower(letra))
       putchar(letra);
     else
       {
         if (isupper(letra))
           {
             ungetch(letra);
             maiusc_encontrado = 1;
             putchar('\n');
           }
         }
         feito = 1;
   while (! feito);

   if (maiusc_encontrado)
     do
      {
         letra = getch();
     .}  putchar(letra);
     while (letra != '\r');
 }
```

Se você estiver lendo caracteres usando *getchar*, poderá usar função *ungetc* para anular a leitura de um caractere, como mostrado aqui:

```
ungetc(letra, stdin);
```

SAÍDA FORMATADA RÁPIDA USANDO CPRINTF 297

Como você sabe, a função *printf* permite que seus programas efetuem saída formatada. C na verdade define a função *printf* em termos do indicat.vo (*handle*) de arquivo *stdout*. Como resultado, você pode redirecionar a saída de *printf* da tela para um arquivo ou dispositivo. Como *printf* usa *stdout* para exibir caracteres, *printf* usa o sistema de arquivo de C, que, por sua vez, usa as funções do DOS. Cada uma das funções do DOS, por sua vez, chama a BIOS. Para saída formatada mais rápida, seus programas podem usar a seguinte função, *cprintf*, que trabalha diretamente com a BIOS ou a memória de vídeo do seu computador:

```
#include <conio.h>

int cprinft(const char *formato[,argumentos...]);
```

O seguinte programa, *cprintf.c*, escreve a string Programando em C/C++ — A Bíblia 1.001 vezes na sua tela usando *printf* e *daí cprintf*. O programa então exibe um sumário do tempo requerido para cada função:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
 {
   int conta;
```

```
    time_t hora_inicio, hora_parada;
    time(&hora_inicio);
    for (conta = 0; conta < 1001; conta++)
       printf("Bíblia do Programador C/C++\n");
    time(&hora_parada);
    printf("\n\nTempo requerido por printf %d segundos\n", hora_parada
    -hora_inicio);
    printf("Pressione qualquer tecla...\n");
    getch();
    time(&hora_inicio);
    for (conta = 0; conta < 1001; conta++)
      cprintf("Bíblia do Programador C/C++\r\n");
    time(&hora_parada);
    printf("\n\nTempo requerido por cprintf %d segundos\n",
      hora_parada-hora_inicio);
  }
```

Nota: A função cprintf não converte o caractere de nova linha para uma seqüência de retorno do carro e alimentação de linha.

298 ENTRADA RÁPIDA FORMATADA A PARTIR DO TECLADO

Na dica anterior você aprendeu que *cprintf* permite que seus programas ignorem o sistema de arquivos para efetuar saída rápida na tela. De um modo similar, a função *cscanf* permite que seus programas executem entrada formatada rápida a partir do teclado, como mostrado aqui:

```
#include <conio.h>

int cscanf(char *formato[,argumentos]);
```

O programa a seguir, *cscanf.c* lhe pede três valores inteiros. O programa então lê os valores usando *cscanf*:

```
#include <conio.h>

void main(void)
 {
   int a, b, c;

   cprintf("Digite 3 valores inteiros e pressione Enter\r\n");
   cscanf("%d %d %d", &a, &b, &c);
   cprintf("Os valores digitados foram %d %d %d\r\n", a, b, c);
 }
```

299 ESCREVENDO UMA STRING DE CARACTERES

Como você aprendeu, a função *printf* permite que seus programas escrevam saída formatada na tela. Usando *printf*, seus programas podem escrever strings, inteiros, números em ponto flutuante, ou combinações de diferentes valores na tela. No entanto, quando seus programas somente precisam escrever uma string de caractere, você pode estar apto a aumentar o desempenho do seu programa usando a função *puts* em vez de *printf*, como mostrado aqui:

```
#include <stdio.h>

int puts(const char *string);
```

A função *puts* escreve uma string terminada por *NULL* na tela (na verdade, em *stdout*). Se *puts* for bem-sucedida, *puts* retorna um valor não-negativo. Se um erro ocorrer, *puts* retorna *EOF*. A função *puts* escreve automaticamente um caractere de nova linha no final da string. O programa a seguir, *puts.c*, usa *printf* e *puts* para

escrever a string "Bíblia do Programador C/C++!" 1.001 vezes. O programa exibe o tempo requerido por cada função:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
 {
   int conta;

   time_t hora_inicio, hora_parada;
   time(&hora_inicio);
   for (conta = 0; conta < 1001; conta++)
      printf("Bíblia do Programador C/C++\n");
   time(&hora_parada);
   printf("\n\nTempo requerido por printf %d segundos\n", hora_parada
-hora_inicio);
   printf("Pressione qualquer tecla...\n");
   getch();
   time(&hora_inicio);
   for (conta = 0; conta < 1001; conta++)
       puts("Bíblia do Programador C/C++");
   time(&hora_parada);
   printf("\n\nTempo requerido por puts %d segundos\n", hora_parada-
hora_inicio);
 }
```

Nota: Como a função puts acrescenta automaticamente um caractere nova linha, a string de caractere que o programa instrui puts a exibir não inclui o caractere de nova linha.

ESCRITA MAIS RÁPIDA DE STRING USANDO E/S DIRETA 300

Na dica anterior foi visto que a função *puts* permite que seus programas escrevam rapidamente uma string de caracteres. No entanto, como C define a função *puts* em termos de *stdout* (para poder suportar o redirecionamento), a função precisa usar o sistema de arquivos. Para saída mais rápida das strings na tela, seus programas podem querer usar a função *cputs*, como mostrado aqui:

```
#include <conio.h>

int cputs(const char string);
```

Como *puts*, a função *cputs* escreve uma string terminada por *NULL*. No entanto, ao contrário de *puts*, *cputs* não acrescenta automaticamente um caractere de nova linha. O programa a seguir, *cputs.c*, usa as funções *puts* e *cputs* para exibir a string "Bíblia do Programador C/C++" 1.500 vezes. O programa exibe a quantidade de tempo que cada função teve de requerer para gerar a saída:

```
#include <stdio.h>
#include <conio.h>
#include <time.h>

void main(void)
 {
   int conta;

   time_t hora_inicio, hora_parada;
   time(&hora_inicio);
   for (conta = 0; conta < 1500; conta++)
       puts("Bíblia do Programador C/C++");
```

```
    time(&hora_parada);
    printf("\n\nTempo requerido por puts %d segundos\n", hora_parada
    -hora_inicio);
    printf("Pressione qualquer tecla...\n");
    getch();
    time(&hora_inicio);
    for (conta = 0; conta < 1500; conta++)
        cputs("Bíblia do Programador C/C++\r\n");
    time(&hora_parada);
    printf("\n\nTempo requerido por cputs %d segundos\n", hora_parada
    -hora_inicio);
}
```

301 LENDO UMA STRING DE CARACTERES DO TECLADO

Na Dica 299 você aprendeu que C fornece a função *puts*, que você pode usar dentro de seus programas para escrever uma string de caracteres na tela. De um modo similar, seus programas podem usar a função *gets* para ler uma string de caracteres do teclado, como mostrado aqui:

```
#include <stdio.h>

char *gets(char *string);
```

Se for bem-sucedida, *gets* retornará um ponteiro para a string de caracteres. Se um erro ocorrer ou se *gets* encontrar o final do marcador de arquivo, *gets* retornará o valor *NULL*. A função *gets* lê caracteres até e incluindo o caractere de *nova linha*. No entanto, *gets* substitui o caractere de *nova linha* por *NULL*. O programa a seguir, *gets.c*, usa a função *gets* para ler uma string de caracteres do teclado:

```
#include <stdio.h>

void main(void)
 {
   char string[256];

   printf("Digite uma string e pressione Enter\n");
   gets(string);
   printf("A string foi %s\n", string);
 }
```

Nota: C na verdade define a função gets em termos de stdin (que, por padrão, é o teclado), o que permite que a função suporte o redirecionamento da E/S.

302 EXECUTANDO ENTRADA DE STRING MAIS RÁPIDA PELO TECLADO

Na dica anterior você aprendeu como seus programas podem usar a função *gets* para ler uma string de caracteres do teclado. Como C define *gets* em termos de *stdin*, *gets* precisa usar o sistema de arquivos para efetuar suas operações de entrada. Se você não precisar suportar o redirecionamento da E/S, poderá usar a função *cgets* para ler caracteres do teclado, dessa forma aumentando o desempenho do seu programa. Você implementará *cgets* como mostrado aqui:

```
#include <conio.h>

char *cgets(char *string);
```

Se *cgets* ler caracteres do teclado com sucesso, retornará um ponteiro para a string que inicia em *string[2]*. Se um erro ocorrer, *cgets* retornará *NULL*. A função *cgets* se comporta de forma diferente que *gets*. Antes de chamar *cgets* com uma string de caracteres, você precisará primeiro atribuir para *string[0]* o número máximo de caractere que *cgets* irá ler. Quando *cgets* retornar, *string[1]* conterá um contador de caracteres que *cgets* leu. A string de caracteres terminada por *NULL* na verdade inicia em *string[2]*. O programa a seguir, *cgets.c*, ilustra como usar a função *cgets*:

```
#include <stdio.h>
#include <conio.h>

void main(void)
 {
   char buffer[256];

   buffer[0] = 253; // Número de caracteres que podem ser lidos
   printf("Digite uma string e pressione Enter\n");
   cgets(buffer);
   printf("\n\nO número de caracteres lidos foi %d\n", buffer[1]);
   printf("String lida: %s\n", &buffer[2]);
 }
```

Para experimentar este programa, reduza o número de caracteres que *cgets* pode ler para 10. Se o usuário tentar digitar mais do que 10 caracteres, a função ignorará os caracteres extras.

EXIBINDO SAÍDA EM CORES 303

Usando o controlador de dispositivo *ansi.sys*, seus programas podem exibir saída na tela em cores. Além disso, muitos compiladores fornecem funções de saída baseadas em texto que lhe permitem exibir saída em cores. Se você estiver usando o *Turbo C++ Lite*, o Borland C++ ou o Microsoft C++, a função *outtext* (chamada *_outtext* no Microsoft C++) lhe permitirá exibir saída em cores. Se você estiver usando o *Turbo C++ Lite* ou o Borland C++, somente poderá usar a função *outtext* no modo gráfico. A função *_outtext* da Microsoft, por outro lado, trabalha no modo texto e no modo gráfico. Se você precisar escrever saída em cores, consulte a documentação do seu compilador para conhecer os detalhes específicos sobre essas funções. Como você verá, os compiladores fornecem funções que definem a posição do texto, as cores e os modos gráficos. Como as rotinas de saída ANSI são dependentes do compilador, este livro não discutirá essas rotinas como dicas.

LIMPANDO A TELA DO VÍDEO 304

A maioria dos compiladores C não fornece uma função que permita limpar a tela. No entanto, caso esteja usando o *Turbo C++ Lite*, o Borland C ou o Microsoft C, você pode usar a função *clrscr* para limpar o conteúdo de uma janela modo texto, como mostrado aqui:

```
#include <conio.h>

void clrscr(void);
```

O programa a seguir, *clrscr.c*, usa a função *clrscr* para limpar a tela de vídeo:

```
#include <conio.h>
void main(void)
 {
   clrscr;
 }
```

APAGANDO ATÉ O FINAL DA LINHA ATUAL 305

À medida que seus programas forem efetuando E/S na tela, algumas vezes você poderá querer apagar o conteúdo de uma linha da posição atual do cursor até o final da linha. Para fazer isso, seus programas podem usar a função *clreol*, como mostrado aqui:

```
#include <conio.h>

void clreol(void);
```

A função *clreol* apaga o conteúdo da linha atual a partir da posição do cursor, sem mover o cursor.

306 APAGANDO A LINHA ATUAL DA TELA

À medida que seus programas efetuarem E/S baseada na tela, algumas vezes você desejará apagar o conteúdo da linha atual, movendo toda a saída que segue uma linha para cima. Em tais casos, seus programas podem usar a função *delline*, como mostrado aqui:

```
#include <conio.h>

void delline(void);
```

O programa a seguir, *delline.c*, preenche a tela com 24 linhas de texto. Quando você pressionar Enter, o programa usará *delline* para apagar as linhas 12, 13 e 14, como mostrado aqui:

```
#include <conio.h>

void main(void)
 {
   int linha;

   clrscr();
   for (linha = 1; linha < 25; linha++)
      cprintf("Esta é a linha %d\r\n", linha);
   cprintf("Pressione uma tecla para continuar: ");
   getch();
   gotoxy(1, 12);
   for (linha = 12; linha < 15; linha++)
        delline();
   gotoxy(1, 25);
 }
```

307 POSICIONANDO O CURSOR PARA SAÍDA NA TELA

Como você sabe, é possível usar o controlador de dispositivo *ansi.sys* para posicionar o cursor para as operações de escrita na tela. Se você estiver trabalhando no ambiente do DOS, muitos compiladores fornecerão a função *gotoxy*, que lhe permitirá posicionar o cursor em uma interseção específica de linha e coluna, como mostrado aqui:

```
#include <conio.h>

void gotoxy(int coluna, int linha);
```

O parâmetro *coluna* especifica uma posição de coluna (x) de 1 até 80. O parâmetro *linha* especifica uma posição de linha (y) de 1 até 25. Se um dos valores for inválido, o compilador ignorará a operação *gotoxy*. O seguinte programa, *gotoxy.c*, usa a função *gotoxy* para exibir saída na tela em posições específicas:

```
#include <conio.h>

void main(void)
 {
   clrscr();
   gotoxy(1, 5);
   cprintf("Saída na linha 5 coluna 1\n");
   gotoxy(20, 10);
   cprintf("Saída na linha 10 coluna 20\n");
 }
```

DETERMINANDO A POSIÇÃO DE LINHA E DE COLUNA 308

Na Dica 307 você aprendeu como usar *gotoxy* para colocar o cursor em uma posição específica de linha e coluna. Em muitos casos, seus programas poderão querer conhecer a posição atual do cursor antes de executar uma operação de E/S na tela. As funções *wherex* e *wherey* retornam a coluna e a linha do cursor, como mostrado aqui:

```
#include <conio.h>

int wherex(void);
int wherey(void);
```

O programa a seguir, *wherexy.c*, limpa a tela, escreve três linhas de saída, e, depois, usa as funções *wherex* e *wherey* para determinar a posição atual do cursor:

```
#include <conio.h>

void main(void)
 {
   int linha, coluna;

   clrscr();
   cprintf("Esta é a linha 1\r\n");
   cprintf("A linha 2 é um pouco mais longa\r\n");
   cprintf("Esta é a última linha");
   linha = wherey();
   coluna = wherex();
   cprintf("\r\nA posição do cursor era lin %d col %d\n", linha, coluna);
 }
```

INSERINDO UMA LINHA EM BRANCO NA TELA 309

À medida que seus programas efetuarem operações de E/S baseadas na tela, algumas vezes você desejará inserir uma linha em branco na tela para poder inserir texto no meio de texto existente. Para fazer isso, seus programas podem usar a função *insline*, como mostrado aqui:

```
#include <conio.h>

void insline(void);
```

Quando você chama a função *insline*, todo o texto abaixo da posição atual do cursor move-se para baixo da linha. A linha na parte inferior da tela rola para fora da janela. O programa a seguir, *insline.c*, escreve 25 linhas de texto na tela. O programa então usa a função *insline* para inserir texto na linha 12, como mostrado aqui:

```
#include <conio.h>

void main(void)
 {
   int linha;

   clrscr();
   for (linha = 1; linha < 25; linha++)
      cprintf("Esta é a linha %d\r\n", linha);
   cprintf("Pressione uma tecla para continuar: ");
   getch();
   gotoxy(1, 12);
   insline();
   cprintf("Isto é novo texto!!!");
   gotoxy(1, 25);
 }
```

310 COPIANDO TEXTO DA TELA PARA UM BUFFER

Quando seus programas efetuam muita E/S na tela, algumas vezes o programa precisa copiar o conteúdo atual da tela para um buffer. Para copiar texto da tela, seus programas podem usar a função *gettext*, como mostrado aqui:

```
#include <conio.h>

int gettext(int esq, int topo, int dir, int base, void *buffer);
```

Os parâmetros *esq* e *topo* especificam as posições de coluna e linha do canto superior esquerdo da região da tela que você quer copiar. Da mesma forma, os parâmetros *dir* e *base* especificam o canto inferior direito da região. A função *gettext* coloca o texto e seus atributos no parâmetro *buffer*. O PC usa um byte de atributo para toda letra de texto que exibe na sua tela. Por exemplo, se você quiser bufferizar 10 caracteres, seu buffer precisará ser grande o suficiente para conter os 10 caracteres ASCII mais os 10 bytes de atributo (20 bytes em comprimento). O programa a seguir, *salvatel.c*, salva o conteúdo da tela em modo texto no arquivo *salvatel.dat*:

```
#include <conio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

void main(void)
 {
   char buffer[8000];
   int indicat;

   if ((indicat = creat("SALVATEL.DAT",S_IWRITE))== -1)
      cprintf("Erro ao abrir SALVATEL.DAT\r\n");
   else
    {
       gettext(1, 1, 80, 25, buffer);
       write(indicat, buffer, sizeof(buffer));
       close(indicat);
    }
 }
```

Nota: Na maioria dos casos, o atributo do texto atual é 7. Se você tentar exibir o conteúdo do arquivo salvatel.dat usando o comando TYPE, seu sistema soará o aviso sonoro para cada byte de atributo.

311 ESCREVENDO UM BUFFER DE TEXTO EM UMA POSIÇÃO ESPECÍFICA DA TELA

Como foi visto, muitos compiladores baseados no DOS fornecem funções que seus programas podem usar para controlar a saída no vídeo. Na dica anterior você aprendeu que seus programas podem usar a função *gettext* para copiar um intervalo de caracteres de tela (e seus atributos) para um buffer. Após você copiar o buffer de texto, poderá depois copiá-lo de volta para a tela usando a função *puttext*, como mostrado aqui:

```
#include <conio.h>

int puttext(int esq, int topo, int dir, int base, void *buffer);
```

Os parâmetros *esq*, *topo*, *dir* e *base* especificam as posições de tela nas quais você quer que o conteúdo do buffer seja escrito. O parâmetro *buffer* contém os caracteres e atributos que *gettext* armazenou anteriormente. O programa a seguir, *puttext.c*, move o texto "Bíblia do Programador C/C++" na sua tela até que você pressione uma tecla qualquer

```
#include <conio.h>
#include <io.h>
```

```
#include <fcntl.h>
#include <sys\stat.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
 {
   char buffer[128];
   int linha, coluna;

   clrscr();
   cprintf("Bíblia do Programador C/C++\r\n");
   gettext(1, 1, 23, 1, buffer);
   while (! kbhit())
    {
      clrscr();
      linha = 1 + random(24);
      coluna = 1 + random(58);
      puttext(coluna, linha, coluna+22, linha, buffer);
      delay(2000);
    }
 }
```

DETERMINANDO AS DEFINIÇÕES DO MODO TEXTO 312

Como você aprendeu, muitos compiladores oferecem várias funções baseadas em texto que seus programas podem usar para controlar as operações de escrita na tela. Para determinar as definições atuais da tela, seus programas podem usar a função *gettextinfo*, como mostrado aqui:

```
#include <conio.h>

void gettextinfo(struct text_info *dados);
```

O parâmetro *dados* é um ponteiro para uma estrutura do tipo *text_info*, como mostrado aqui:

```
struct text_info
 {
   unsigned char winleft;          // coluna da esquerda
   unsigned char wintop;           // linha superior
   unsigned char winright;         // coluna da direita
   unsigned char winbottom;        // linha inferior
   unsigned char attribute;        // atributo do texto
   unsigned char normattr;         // atributo normal
   unsigned char currmode;         // modo atual do texto
   unsigned char screenheight;     // altura em linhas
   unsigned char screenwidth;      // largura em colunas
   unsigned char curx;             // coluna do cursor
   unsigned char cury;             // linha do cursor;
 }
```

O programa a seguir, *textinfo.c*, usa a função *gettextinfo* para exibir a definição de texto atual:

```
#include <conio.h>

void main(void)
 {
   struct text_info texto;

   gettextinfo(&texto);
   cprintf("Coordenadas da tela %d,%d até %d,%d\r\n",
       texto.wintop, texto.winleft, texto.winbottom, texto.winright);
   cprintf("Atributos do texto %d Atributo normal %d\r\n",  texto.
```

```
attribute, texto.normattr);
   cprintf("Altura da tela %d largura %d\r\n", texto.screenheight,
   texto.screenwidth);
   cprintf("A posição do cursor era lin %d col %d\r\n", texto.cury,
   texto.curx);
 }
```

313 CONTROLANDO AS CORES DA TELA

Já vimos que seus programas podem usar o controlador de dispositivo *ansi.sys* para exibir saída colorida na tela. Além disso, muitos compiladores baseados no DOS fornecem a funçao *textattr*, que lhe permite selecionar as cores de frente e de fundo do texto:

```
#include <conio.h>

void textattr(int atributo);
```

O parâmetro *atributo* contém oito bits que especificam as cores que você deseja. Os quatro bits menos significativos especificam a cor da frente. Os três bits seguintes especificam a cor de fundo, e o bit mais significativo controla a intermitência. Para selecionar uma cor, você precisa atribuir o valor da cor desejada para os bits corretos. A Tabela 313 especifica os valores das cores.

Tabela 313 Os parâmetros de atributos de cores.

Constante de Cor	Valor	Uso
BLACK (Preto)	0	Frente/fundo
BLUE (Azul)	1	Frente/fundo
GREEN (Verde)	2	Frente/fundo
CYAN (Ciano)	3	Frente/fundo
RED (Vermelho)	4	Frente/fundo
MAGENTA (Magenta)	5	Frente/fundo
BROWN (Marrom)	6	Frente/fundo
LIGHTGRAY (Cinza claro)	7	Frente/fundo
DARKGRAY (Cinza escuro)	8	Frente
LIGHTBLUE (Azul claro)	9	Frente
LIGHTGREEN (Verde claro)	10	Frente
LIGHTCYAN (Ciano claro)	11	Frente
LIGHTRED (Vermelho claro)	12	Frente
LIGHTMAGENTA (Magenta claro)	13	Frente
YELLOW (Amarelo)	14	Frente
WHITE (Branco)	15	Frente
BLINK (Piscante)	128	Frente

O programa a seguir, *textattr.c*, ilustra as cores de frente disponíveis:

```
#include <conio.h>

void main(void)
 {
   int cor;

   for (cor = 1; cor < 16; cor++)
    {
      textattr(cor);
      cprintf("Esta é a cor %d\r\n", cor);
    }
   textattr(128 + 15);
   cprintf("Isto é piscante \r\n");
 }
```

ATRIBUINDO CORES DE FUNDO 314

Como você aprendeu na Dica 313, a função *textattr* permite que seus programas selecionem cores de frente e de fundo. Para definir a cor de fundo usando *textattr*, seu programa precisa atribuir o valor da cor que você deseja nos bits de 4 a 6 do valor da cor. Para atribuir o valor da cor, seus programas podem usar operações de deslocamento bit a bit, ou você pode declarar uma estrutura com campos de bit, como mostrado aqui:

```
struct TextColor {
   unsigned char foreground:4;
   unsigned char background:3;
   unsigned char blinking:1;
};
```

O programa a seguir, *defcores.c*, usa a estrutura *TextColor* para definir as cores atuais da tela:

```
#include <conio.h>

void main(void)
 {
   union TextColor
    {
       struct
        {
           unsigned char frente:4;
           unsigned char fundo:3;
           unsigned char piscante:1;
        } bits_cores;
       unsigned char valor;
    } cores;
   cores.bits_cores.frente = BLUE;
   cores.bits_cores.fundo = RED;
   cores.bits_cores.piscante = 1;
   textattr(cores.valor);
   clrscr();
   cprintf("Esta é a nova cor do texto \n");
 }
```

DEFININDO A COR DE FRENTE USANDO TEXTCOLOR 315

Como você aprendeu, muitos compiladores baseados no DOS fornecem a função *textattr*, que lhe permite selecionar suas cores de frente e de fundo. Para simplificar o processo de atribuir uma cor de frente, você pode querer usar a função *textcolor*, como mostrado aqui:

```
#include <conio.h>

void textcolor(int corfrente);
```

O parâmetro *corfrente* especificará um dos valores de cores listados na Tabela 315.

Tabela 315 Valores de cores de frente válidos para ***textcolor***.

Constante de Cor	Valor	Constante de Cor	Valor
BLACK (Preto)	0	*DARKGRAY* (Cinza escuro)	8
BLUE (Azul)	1	*LIGHTBLUE* (Azul claro)	9
GREEN (Verde)	2	*LIGHTGREEN* (Verde claro)	10
CYAN (Ciano)	3	*LIGHTCYAN* (Ciano claro)	11
RED (Vermelho)	4	*LIGHTRED* (Vermelho claro)	12
MAGENTA (Magenta)	5	*LIGHTMAGENTA* (Magenta claro)	13
BROWN (Marrom)	6	*YELLOW* (Amarelo)	14
LIGHTGRAY (Cinza claro)	7	*WHITE* (Branco)	15
		BLINK (Piscante)	128

O programa a seguir, *textcolor.c*, ilustra como usar a função *textcolor* para definir a cor de frente:

```
#include <conio.h>

void main(void)
 {
   int cor;

   for (cor = 1; cor < 16; cor++)
    {
       textcolor(cor);
       cprintf("Esta é a cor %d\r\n", cor);
    }
   textcolor(128 + 15);
   cprintf("Isto é piscante\r\n");
 }
```

316 DEFININDO A COR DE FUNDO USANDO TEXTBACKGROUND

Já vimos que muitos compiladores baseados no DOS fornecem a função textattr, que lhe permite selecionar as cores de frente e de fundo que você quer para a exibição do texto. Para simplificar o processo de atribuir uma cor de fundo, você pode usar a função *textbackground*, como mostrado aqui:

```
#include <conio.h>

void textbackground(int corfundo);
```

O parâmetro *backgroundcolor* especificará um dos valores de cores listados na Tabela 316:

Tabela 316 Valores válidos de cores de fundo.

Constante de Cor	Valor	Constante de Cor	Valor
BLACK (Preto)	0	*RED* (Vermelho)	4
BLUE (Azul)	1	*MAGENTA* (Magenta)	5
GREEN (Verde)	2	*BROWN* (Marrom)	6
CYAN (Ciano)	3	*LIGHTGRAY* (Cinza claro)	7

O programa a seguir, *fundo.c*, usa a função *textbackground* para exibir as diferentes cores de fundo:

```
#include <conio.h>

void main(void)
 {
  int cor;

  for (cor = 0; cor < 8; cor++)
  {
     textbackground(cor);
     cprintf("Esta é a cor %d\r\n", cor);
     cprintf("Pressione qualquer tecla para continuar\r\n");
     getch();
  }
 }
```

317 CONTROLANDO A INTENSIDADE DO TEXTO

Como você aprendeu, muitos compiladores baseados no DOS fornecem funções que lhe permitem controlar a saída na tela. Ao usar essas funções para escrever texto na tela, você irá querer controlar a intensidade (brilho) das informações que seus programas escrevem na tela. Para controlar a intensidade, você poderá usar uma das três funções a seguir para selecionar a intensidade da saída do texto:

```
#include <conio.h>

void highvideo(void);
void lowvideo(void);
void normvideo(void);
```

As funções controlam a intensidade com a qual sua tela exibirá texto. O programa a seguir, *intensid.c*, ilustra como usar essas três funções:

```
#include <conio.h>

void main(void)
 {
   clrscr();
   highvideo();
   cprintf("Este texto está em alta intensidade\r\n");
   lowvideo();
   cprintf("Este texto está em baixa intensidade\r\n");
   normvideo();
   cprintf("Este texto está em vídeo normal\r\n");
 }
```

DETERMINANDO O MODO ATUAL DO TEXTO 318

Como você sabe, muitos compiladores baseados no DOS fornecem funções que seus programas podem usar para controlar a saída baseada em texto. Quando seus programas efetuam saída na tela, eles precisam conhecer e possivelmente alterar o modo de texto atual do PC. Por exemplo, um programa que espera 80 colunas exibirá resultados inconsistentes em uma tela que esteja no modo de 40 colunas. Para ajudar seus programas a alterar o modo de texto atual, seus programas podem usar a função *textmode*, como mostrado aqui:

```
#include <conio.h>

void textmode(int modo_desejado);
```

O parâmetro *modo_desejado* especifica o modo de texto que você deseja. A Tabela 318 lista os modos de textos válidos.

Tabela 318 Operações válidas do modo de texto.

Constante	Valor	Modo de Texto
LASTMODE	-1	Modo anterior
BW40	0	Branco e preto em 40 colunas
C40	1	40 colunas em cores
BW80	2	80 colunas, branco e preto
C80	3	80 colunas, colorido
MONO	7	80 colunas, monocromático
C4350	64	EGA de 43 linhas ou VGA de 50 linhas

Por exemplo, o comando a seguir selecionará um modo de 43 linhas em um monitor EGA ou modo de 50 linhas em um monitor VGA:

```
textmode(C4350);
```

Nota: Se você usar textmode para alterar o modo atual da tela, a mudança permanecerá em efeito após seu programa terminar.

MOVENDO TEXTO DA TELA DE UM LOCAL PARA OUTRO 319

Você aprendeu que muitos compiladores baseados no DOS oferecem funções que lhe permitem controlar a saída do texto. Se seu programa realizar muita saída na tela, algumas vezes você poderá querer copiar ou mover o texto

que aparece em uma seção da sua tela para outra seção. Para copiar texto na tela, seus programas podem usar a função *movetext*, como mostrado aqui:

```
#include <conio.h>

int movetext(int esq, int topo, int dir, int base,
      int destino_esq, int destino_topo);
```

Os parâmetros *esq*, *topo*, *dir* e *base* descrevem uma caixa que delimita a região de texto que você quer mover. Os parâmetros *destino_esq* e *destino_topo* especificam a localização desejada do canto superior esquerdo da caixa. O programa a seguir, *movetext.c*, escreve cinco linhas de texto na tela, e, depois, pede para você pressionar uma tecla. Ao fazer isso, o programa copiará o texto para uma nova posição, como mostrado aqui:

```
#include <conio.h>

void main(void)
 {
   int i;

   clrscr();
   for (i = 1; i <= 5; i++)
      cprintf("Esta é a linha %d\r\n", i);
   cprintf("Pressione qualquer tecla\n\r");
   getch();
   movetext(1, 1, 30, 6, 45, 18);
   gotoxy(1, 24);
 }
```

Para mover o texto para a nova posição, em vez de apenas copiar o texto para a nova posição, você precisa excluir o texto original após o programa completar a operação *movetext*.

320 DEFININDO UMA JANELA DE TEXTO

Sabemos que muitos compiladores com base no DOS fornecem funções que seus programas podem usar para controlar melhor a saída na tela. Por padrão, essas funções escrevem sua saída na tela inteira. Dependendo do propósito do seu programa, algumas vezes você desejará restringir a saída do programa para uma região específica na tela. Para fazer isso, seus programas podem usar a função *window*, como mostrado aqui:

```
#include <conio.h>

void window(int esq, int topo, int dir, int base);
```

Os parâmetros *esq*, *topo*, *dir* e *base* definem os cantos superior esquerdo e inferior direito de uma região de tela dentro da qual você quer escrever a saída. O programa a seguir, *janela.c*, restringe a saída do programa ao quadrante superior esquerdo da tela:

```
#include <conio.h>

void main(void)
 {
   int i, j;

   window(1, 1, 40, 12);
   for (i = 0; i < 15; i++)
    {
      for (j = 0; j < 50; j++)
        cprintf("%d", j);
      cprintf("\r\n");
    }
 }
```

Quando a saída do programa atinge o canto direito da tela, a saída quebra para a próxima linha. Após o programa terminar, as operações de saída terão acesso à tela inteira.

USANDO O VALOR ABSOLUTO DE UMA EXPRESSÃO INTEIRA 321

O *valor absoluto* especifica a distância do valor de 0. Os valores absolutos são sempre positivos. Por exemplo, o valor absoluto de 5 é 5. Da mesma forma, o valor absoluto de -5 é 5. Para ajudar seus programas a determinar um valor absoluto, C fornece a função *abs*, que retorna o valor absoluto de uma expressão inteira. Você construirá a função *abs* como mostrado aqui:

```
#include <stdlib.h>

int abs(int expressão);
```

O programa a seguir, *exib_abs.c*, ilustra como usar a função *abs*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   printf("O valor absoluto de %d é %d\n", 5, abs(5));
   printf("O valor absoluto de %d é %d\n", 0, abs(0));
   printf("O valor absoluto de %d é %d\n", -5, abs(-5));
 }
```

Quando você compilar e executar o programa *exib_abs.c*, sua tela exibirá o seguinte:

```
O valor absoluto de 5 é 5
O valor absoluto de 0 é 0
O valor absoluto de -5 é 5
C:\>
```

Nota: Muitos compiladores C também fornecem a labs, que retorna o valor absoluto para uma expressão do tipo long int.

USANDO O ARCO CO-SENO 322

O arco co-seno é a relação entre a hipotenusa de um triângulo retângulo e o cateto adjacente de um determinado ângulo agudo. Em outras palavras, o arco co-seno é o inverso geométrico do co-seno de um ângulo. Em outras palavras, se *y* é o co-seno de algum ângulo teta, teta é o arco co-seno de *y*. Para ajudar seus programas a determinar o arco co-seno, C fornece a função *acos*. A função *acos* retorna o arco co-seno de um ângulo (de 0 a pi) especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double acos(double expressão);
```

Se a expressão especificada não estiver no intervalo de -1.0 até 1.0, *acos* definirá a variável global *errno* para *EDOM* e exibirá um erro *DOMAIN* em *stderr*. O programa a seguir, *exi_acos.c*, ilustra como usar a função *acos*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double radianos;

   for (radianos= -0.5; radianos <= 0.5; radianos + = 0.2)
      printf("%f %f\n", radianos, acos(radianos));
 }
```

Nota: Muitos compiladores C também fornecem a função acosl, que retorna o valor arco co-seno para uma expressão long double.

323 USANDO O ARCO SENO

O arco seno é a relação entre a hipotenusa de um triângulo retângulo e o cateto oposto a um determinado ângulo agudo. Em outras palavras, o arco seno é o inverso geométrico do seno de um ângulo. Se *y* é o seno de algum ângulo teta, então teta é o arco seno de *y*. Para ajudar seus programas a determinar o arco seno, C fornece a função *acos*. A função *acos* retorna o arco seno de um ângulo (-pi/2 até pi/2), especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double asin(double expressão);
```

Se expressao nao estiver no intervalo de -1.0 até 1.0, então *asin* definirá a variável global *errno* para *NAN*, e exibirá um erro *DOMAIN* em *stderr*. O programa a seguir, *ex1_asin.c*, ilustra como usar a função *asin*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double radianos;

   for (radianos = -0.5; radianos < =0.5; radianos += 0.2)
     printf("%f %f\n", radianos, asin(radianos));
 }
```

Nota: Muitos compiladores C também fornecem a função asinl, que retorna o valor arco seno para uma expressão long double.

324 USANDO O ARCO TANGENTE

O arco tangente é a relação entre o cateto adjacente a um dado ângulo agudo e o cateto oposto a esse ângulo em um triângulo retângulo. Em outras palavras, o arco tangente é o inverso geométrico da tangente de um ângulo. Se *y* é a tangente de um ângulo teta, teta é o arco tangente de *y*. Para ajudar seus programas a determinar o arco tangente, C fornece a função *atan*. A função *atan* retorna o arco tangente de um ângulo (-pi/2 até pi/2), especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double atan(double expressão);
```

O programa a seguir, *ex1_atan.c*, ilustra como usar a função *atan*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double radianos;

   for (radianos = -0.5; radianos< = 0.5; radianos += 0.2)
      printf("%f %f\n", radianos, atan(radianos));
 }
```

Nota: Muitos compiladores C também fornecem a função atanl, que retorna o valor do arco tangente para uma expressão long double. Além disso, C fornece as funções atan2 e atan2l, que retornam o arco tangente de y/x.

325 OBTENDO O VALOR ABSOLUTO DE UM NÚMERO COMPLEXO

Como você aprendeu, um número complexo contém uma parte real e uma parte imaginária. As funções de C representam os números complexos como uma estrutura com um membro *x* e um membro *y*, como mostrado aqui:

```
struct complex
 {
   double x, y;
 }
```

Quando você trabalhar com números complexos, algumas vezes precisará calcular o valor absoluto do número (sua distância positiva do zero). Para permitir que seu programa calcule o valor absoluto de um número complexo, C fornece a função *cabs*, como mostrado aqui:

```
#include <math.h>

double cabs(struct complex valor);
```

A função *cabs* é similar a tirar a raiz quadrada da soma do quadrado de cada número complexo. No exemplo a seguir, a função *cabs* retornará $(10^2 + 5^2)^{1/2}$. O programa a seguir, *exi_cabs.c*, ilustra como usar a função *cabs* da linguagem C:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   struct complex num_complexo;

   num_complexo.x = 10;
   num_complexo.y = 5;
   printf("O valor absoluto de 10,5 é %f\n", cabs(num_complexo));
 }
```

Quando você compilar e executar o programa *exi_cabs.c*, sua tela exibirá a seguinte saída:

```
O valor absoluto de 10,5 é 11.180340
C:\>
```

Nota: Muitos compiladores C também fornecem a função cabls, que retorna um valor absoluto para números complexos long double.

ARREDONDANDO PARA CIMA UM VALOR EM PONTO FLUTUANTE 326

Quando você trabalha com números em ponto flutuante, algumas vezes precisa arredondar o valor de uma variável em ponto flutuante ou de uma expressão para o valor inteiro mais alto. Para esses casos, C fornece a função *ceil*, como mostrado aqui:

```
#include <math.h>

double ceil(double valor);
```

Como você pode ver, *ceil* recebe um parâmetro do tipo *double* e retorna um valor do tipo *double*. O programa a seguir, *exi-ceil.c*, ilustra como usar a função *ceil*.

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("O valor %f é arredondado para %f\n", 1.9, ceil(1.9));
   printf("O valor %f é arredondado para %f\n", 2.1, ceil(2.1));
 }
```

Quando você compilar e executar o programa *exi_ceil.c*, sua tela exibirá a seguinte saída:

```
O valor 1.900000 é arredondado para 2.00000
O valor 2.100000 é arredondado para 3.00000
C:\>
```

Nota: Muitos compiladores também fornecem a função ceill, que arredonda para cima um valor do tipo long double.

327 USANDO O CO-SENO DE UM ÂNGULO

Em um triângulo retângulo, o co-seno de um ângulo é o quociente do cateto adjacente pela hipotenusa. Para ajudar seus programas a determinar o co-seno, C fornece a função *cos*. A função *cos* retorna o co-seno de um ângulo, especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double cos(double expressão);
```

A função *cos* retorna um valor no intervalo de -1.0 até 1.0. O programa a seguir, *exib_cos.c*, ilustra como usar a função *cos*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("O co-seno de pi/2 é %6.4f\n", cos(3.14159/2.0));
   printf("O co-seno de pi é %6.4f\n",cos(3.14159));
 }
```

Quando você compilar e executar o programa *exib_cos.c*, sua tela exibirá o seguinte resultado:

```
O co-seno de pi/2 é 0.0000
O co-seno de pi é -1.0000
C:\>
```

Nota: Muitos compiladores C também fornecem a função cosl, que retorna o valor do co-seno para uma expressão long double.

328 USANDO O CO-SENO HIPERBÓLICO DE UM ÂNGULO

O co-seno hiperbólico de um ângulo é o co-seno de um ângulo "na forma circular" definido, usando relações de radianos hiperbólicos. Para ajudar seus programas a determinar o co-seno hiperbólico, C fornece a função *cosh*. A função *cosh* retorna um co-seno hiperbólico de um ângulo "circular", especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double cosh(double expressão);
```

Se ocorrer extravasamento, *cosh* retornará o valor *HUGE_VAL* (ou *_LHUGE_VAL* para *cosh*) e definirá a variável global *errno* como *ERANGE*. O programa a seguir, *exi_cosh.c*, ilustra como usar a função *cosh*.

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double radianos;

   for (radianos = -0.5; radianos< = 0.5; radianos + = 0.2)
      printf("%f %f\n", radianos, cosh(radianos));
 }
```

Nota: Muitos compiladores C também fornecem a função coshl, que retorna o valor do co-seno hiperbólico para uma expressão long double.

USANDO O SENO DE UM ÂNGULO 329

Em um triângulo retângulo, o seno de um ângulo é o quociente entre o cateto oposto e a hipotenusa. Para ajudar seus programas a determinar o seno, C fornece a função *sin*. A função *sin* retorna o seno de um ângulo, especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double sin(double expressão);
```

O programa a seguir, *exib_sin.c*, ilustra como usar a função *sin*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
  double radianos;

  for (radianos = 0.0; radianos < 3.1; radianos + = 0.1)
     printf("O seno de %f é %f\n", radianos, sin(radianos));
 }
```

Nota: Muitos compiladores C também fornecem a função sinl, que retorna o valor do seno para uma expressão long double.

USANDO O SENO HIPERBÓLICO DE UM ÂNGULO 330

O seno hiperbólico de um ângulo é o seno de um ângulo "na forma circular" definido, usando relações de radianos hiperbólicos. Para ajudar seus programas a determinar o seno hiperbólico, C fornece a função *sinh*. A função *sinh* retorna um seno hiperbólico "na forma circular", especificado em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double sinh(double expressão);
```

Se ocorrer extravasamento, *sinh* retornará o valor *HUGE_VAL* (ou *_LHUGE_VAL* para *sinhl*), e definirá a variável global *errno* como *ERANGE*. O programa a seguir, *ex_sinh.c*, ilustra como usar a função *sinh*:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main(void)
 {
   double radianos;
   double result;

   for (radianos = 0.0; radianos < 3.1; radianos += 0.1)
     if (((result = sinh(radianos)) == HUGE_VAL) && (errno == ERANGE))
       printf("Erro de extravasamento\n");
     else
       printf("O seno de %f é %f\n", radianos, result);
 }
```

Nota: Muitos compiladores C também fornecem a função sinhl, que retorna o valor seno hiperbólico para uma expressão long double.

331 USANDO A TANGENTE DE UM ÂNGULO

Em um triângulo, a tangente de um ângulo é a relação entre o cateto oposto e o cateto adjacente. Para ajudar seus programas a determinar a tangente, C fornece a função *tan*. A função retorna a tangente de um ângulo, especificada em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double tan(double expressão);
```

O programa a seguir, *exib_tan.c*, ilustra como usar a função *tan*:

```
#include <stdio.h>
#include <math.h>

void main (void)
  {
    double pi = 3,14159265;
    printf ("A tangente de pi é %f\n", tan (pi));
    printf ("A tangente de pi/4 é %f\n", tan (pi/4.0));
 }
```

Quando você compilar e executar o programa *exibe_tan.c*, sua tela exibirá o seguinte resultado:

```
A tangente de pi é - 0.000000
A tangente de pi/4 é 1.000000
C:\>
```

Nota: Muitos compiladores C também fornecem a função ***tanl****, que retorna o valor tangente para uma expressão* ***long double****.*

332 USANDO A TANGENTE HIPERBÓLICA DE UM ÂNGULO

A tangente hiperbólica de um ângulo é a tangente de um ângulo "na forma circular" definida, usando relações de radianos hiperbólicos. Para ajudar seus programas a determinar a tangente hiperbólica, C fornece a função *tanh*. A função *tanh* retorna a tangente hiperbólica de um ângulo, especificada em radianos (como tipo *double*), como mostrado aqui:

```
#include <math.h>

double tanh(double expressão);
```

Nota: Muitos compiladores C também fornecem ***tanhl****, que retorna o valor tangente hiperbólico para uma expressão* ***long double****.*

333 REALIZANDO A DIVISÃO INTEIRA

Como você aprendeu, C fornece operadores de divisão (/) e de módulo (%) que permitem que seus programas efetuem uma divisão para determinar o resto de uma operação de divisão. Similarmente, C fornece a função *div*, que divide um valor numerador por um denominador, retornando uma estrutura do tipo *div_t* que contém o quociente e o resto, como mostrado aqui:

```
struct div_t
 {
   int quot;
   int rem;
 } div_t;
```

A função *div* trabalha com valores inteiros, como mostrado aqui:

```
#include <stdlib.h>

div_t div(int numerador, int denominador);
```

O programa a seguir, *div_rest.c*, ilustra como usar a função *div*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   div_t result;

   result = div(11, 3);
   printf('11 dividido por 3 é %d resto %d\n', result.quot, result.rem);
 }
```

Quando você compilar e executar o programa *div_rest.c*, sua tela exibirá a seguinte saída:

```
11 dividido por 3 é 3 resto 2
C:\>
```

Nota: Muitos compiladores C também fornecem a função ldiv, que retorna o quociente e o resto para valores long.

TRABALHANDO COM EXPONENCIAL 334

Quando seus programas efetuam operações matemáticas complexas, freqüentemente precisam calcular o exponencial de e^x. Nesses casos, seus programas podem usar a função *exp*, que retorna um valor do tipo *double*, como mostrado aqui:

```
#include <math.h>

double exp(double x);
```

O programa a seguir, *exib_exp.c*, ilustra como usar a função *exp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double valor;

   for (valor = 0.0; valor <= 1.0; valor += 0.1)
     printf('exp(%f) é %f\n', valor, exp(valor));
 }
```

Nota: Muitos compiladores C também fornecem a função expl, que trabalha com valores do tipo long double.

USANDO O VALOR ABSOLUTO DE UMA EXPRESSÃO EM PONTO FLUTUANTE 335

Como você aprendeu, o *valor absoluto* especifica a distância de um valor até o zero. Os valores absolutos são sempre positivos. Por exemplo, o valor absoluto de 2.5 é 2.5 Da mesma forma, o valor absoluto de -2.5 é 2.5. Quando você trabalhar com valores absolutos, algumas vezes precisará calcular o valor absoluto de uma expressão em ponto flutuante. Para esses casos, C fornece a função *fabs*, que retorna o valor absoluto para um número em ponto flutuante, como mostrado aqui:

```
#include <math.h>

float fabs(float expressão);
```

O programa a seguir, *exi_fabs.c*, ilustra como usar a função *fabs*:

```
#include <stdio.h>
#include <math.h>

void main(void)
```

```
{
  float valor;

  for (valor = -1.0; valor <= 1.0; valor += 0.1)
     printf("Valor %f fabs %f\n", valor, fabs(valor));
}
```

336 USANDO O RESTO EM PONTO FLUTUANTE

Na Dica 82 você aprendeu como usar o operador módulo (%) de C para obter o resto de uma divisão inteira. Dependendo do seu programa, algumas vezes você desejará conhecer o resto de uma divisão em ponto flutuante. Nesses casos, seus programas podem usar a função *fmod* de C para dividir dois valores em ponto flutuante. A função *fmod* retornará o resto como um valor em ponto flutuante, como mostrado aqui:

```
#include <math.h>

double fmod(double x, double y);
```

Como um exemplo, se você chamar *fmod* com os valores 10.0 e 3.0, *fmod* retornará o valor 1.0 (10 dividido por 3 é 3 e o resto é 1). O programa a seguir, *exi_fmod.c* ilustra como usar a função *fmod*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double numerador = 10.0;
   double denominador = 3.0;

   printf("fmod(10, 3) é %f\n", fmod(numerador, denominador));
 }
```

Quando você compilar e executar o programa *exi_fmod.c*, sua tela exibirá a seguinte saída:

```
fmod(10, 3) é 1.000000
C:\>
```

Nota: Muitos compiladores C também fornecem a função fmodl, que retorna a parte fracionária de um valor long double.

337 USANDO A MANTISSA E O EXPOENTE DE UM VALOR EM PONTO FLUTUANTE

Quando seus programas trabalham com valores em ponto flutuante, o computador armazena os valores usando uma mantissa (cujo valor está entre 0.5 e 1.0) e um expoente, como mostrado na Figura 337.

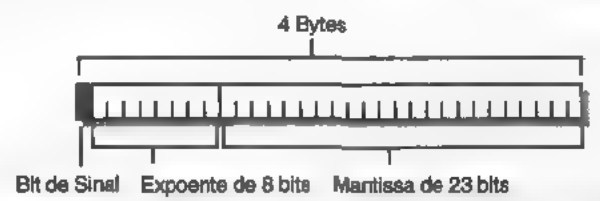

Figura 337 O computador armazena valores em ponto flutuante usando um formato de mantissa e de expoente.

Para determinar o valor armazenado, o computador combina a mantissa e o expoente, como mostrado aqui:

```
valor = mantissa * (2 * expoente);
```

Normalmente, você não precisa saber que o computador está usando a mantissa e o expoente. No entanto, dependendo do seu programa, algumas vezes você poderá querer conhecer os valores da mantissa e do expoente.

Para esses casos, C fornece a função *frexp*, que retorna a mantissa e atribui o expoente à variável *expoente*, que a função chamadora precisa passar para a função *frexp* por referência:

```
#include <math.h>

double frexp(double valor, int *expoente);
```

O programa a seguir, *frexp.c*, ilustra como usar a função *frexp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double valor = 1.2345;
   double mantissa;
   int exponente;

   mantissa = frexp(valor, &expoente);
   printf("Mantissa %f Expoente %d Valor %f\n",
      mantissa, expoente, mantissa * pow(2.0, 1.0 * expoente));
 }
```

Quando você compilar e executar o programa *frexp.c*, sua tela exibirá a seguinte saída:

```
Mantissa 0.617250  Expoente 1  Valor 1.234500
C:\>
```

Nota: Muitos compiladores C também fornecem a função frexpl, que retorna o expoente e a mantissa de um valor long double.

CALCULANDO O RESULTADO DE X * 2E 338

Na Dica 334 você aprendeu como usar a função *exp* de C para obter o resultado *ex*. Dependendo de seus programas, algumas vezes você precisará calcular *x* * 2e. Nessas situações, você poderá usar a função *ldexp*, como mostrado aqui:

```
#include <math.h>

double ldexp(double valor, int expoente);
```

O programa a seguir, *ldexp.c*, ilustra como usar a função *ldexp*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("3 * 2 elevado à potência 4 é %f\n", ldexp(3.0, 4));
 }
```

Quando você compilar e executar o programa *ldexp.c*, sua tela exibirá a seguinte saída.

```
3 * 2 elevado à potência 4 é 48.000000
C:\>
```

Nota: Muitos compiladores C também fornecem a função ldexpl para suportar valores long double.

CALCULANDO O LOGARITMO NATURAL 339

O logaritmo natural de um número é a potência a qual *e* precisa ser elevado para ser igual ao número dado. Para ajudar seus programas a determinar o "logaritmo natural", C fornece a função *log*, que retorna o logaritmo natural de um valor em ponto flutuante:

```
#include <math.h>

double log(double valor);
```

Se o parâmetro *valor* for menor que 0, *log* definirá a variável global *errno* como *ERANGE*, e retornará o valor *HUGE_VAL* (ou *_LHUGE_VAL* para *logl*). O programa a seguir, *exib_log.c*, ilustra o uso da função *log*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("O logaritmo natural de 256.0 é %f\n", log(256.0));
 }
```

Quando você compilar e executar o programa *exib_log.c*, sua tela exibirá a seguinte saída:

```
O logaritmo natural de 256.0 é 5.545177
C:\>
```

Nota: Muitos compiladores também fornecem a função **logl**, *que retorna o logaritmo natural de uma expressão* **long double**.

340 CALCULANDO O RESULTADO DE LOG10X

Você viu na dica anterior como usar a função *log* para calcular um logaritmo natural. À medida que seus programas efetuarem operações matemáticas, algumas vezes você precisará determinar o logaritmo de base 10 de um valor (comumente escrito como *log10x*). Para esses casos, C fornece a função *log10*, como mostrado aqui:

```
#include <math.h>

double log10(double valor);
```

Se o parâmetro *valor* for 0, *log10* definirá a variável global *errno* com *EDOM*, e retornará o valor *HUGE_VAL* (ou *_LHUGE_VAL* para *log10l*). O programa a seguir, *log_10.c*, ilustra como usar a função *log10*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("O log10 de 100 é %f\n", log10(100.0));
   printf("O log10 de 10000 é %f\n", log10(10000.0));
 }
```

Quando você compilar e executar o programa *log_10.c*, sua tela exibirá a seguinte saída:

```
Log10 de 100 é 2.000000
Log10 de 10000 é 4.000000
C:\>
```

Nota: Muitos compiladores C também fornecem a função **log10l**, *que suporta valores* **long double**.

341 DETERMINANDO OS VALORES MÁXIMO E MÍNIMO

Quando seus programas comparam dois números, algumas vezes você deseja conhecer o máximo e o mínimo dentre dois valores. Para esses casos, o arquivo de cabeçalho *stdlib.h* fornece as macros *min* e *max*. O programa a seguir, *min_max.c*, ilustra como usar essas duas macros:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
```

```
    printf("O máximo dentre %f e %f é %f\n", 10.0, 25.0, max(10.0, 25.0));
    printf("O mínimo dentre %f e %f é %f\n", 10.0, 25.0, min(10.0, 25.0));
}
```

Para compreender melhor essas duas macros, considere a seguinte implementação:[1]

```
#define max(x,y) (((x) > (y)) ? (x) : (y))
#define min(x,y) (((x) < (y)) ? (x) : (y))
```

QUEBRANDO UM DOUBLE EM SEUS COMPONENTES INTEIRO E REAL 342

Como você aprendeu, um valor em ponto flutuante consiste de duas partes, uma inteira e uma fracionária. Por exemplo, dado o número 12.345, o valor 12 é a parte inteira e 0.345 é a parte fracionária. Dependendo do seu programa, algumas vezes você desejará trabalhar com os componentes inteiro e fracionário de um valor, ou com cada componente individualmente. Para esses casos, C fornece a função *modf*, como mostrado aqui:

```
#include <math.h>

double modf(double valor, double *parte_inteira);
```

A função *modf* retorna a parte fracionária de um valor e atribui a parte inteira à variável especificada. O programa a seguir, *int_frac.c*, ilustra como usar a função *modf*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double valor = 1.2345;
   double int_part;
   double fracao;

   fracao = modf(valor, &int_part);

   printf("Valor %f  Parte inteira %f  Fracao %f\n", valor, int_part, fracao);
 }
```

Quando você compilar e executar o programa *int_frac.c*, sua tela exibirá a seguinte saída:

```
Valor 1.234500  Parte inteira 1.000000  Fração 0.234500
C:\>
```

Nota: Muitos compiladores C também fornecem a função ***modfl****, que retorna as partes inteira e fracionária de uma expressão* ***long double****.*

CALCULANDO O RESULTADO DE X^N 343

Elevar um valor a uma determinada potência é uma das operações matemáticas mais comuns que seus programas executarão. C fornece a função *pow*, que retorna o resultado de um valor elevado a uma dada potência, como mostrado aqui:

```
#include <math.h>

double pow(double valor, double potencia);
```

Se avaliar o *valor* elevado à dada *potência* resultar em extravasamento, *pow* atribuirá à variável global *errno* o valor *ERANGE*, e retornará *HUGE_VAL* (ou *_LHUGE_VAL* para *powl*) para a função chamadora. Se a função chamadora passar um parâmetro *valor* para *pow*, que é menor que 0, e a potência não for um número inteiro, então *pow* definirá a variável global *errno* com *EDOM*. O programa a seguir, *exib_pow.c*, ilustra como usar a função *pow* de C:

1. N.R.T.: Se seu compilador não definir as macros *min* e *max*, você precisará incluir ambas as declarações *#define* no início do seu arquivo-fonte.

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   int potencia;

   for (potencia = -2; potencia <= 2; potencia++)
      printf("10 elevado a %d é %f\n", potencia, pow(10.0, potencia));
 }
```

Quando você compilar e executar o programa *exib_pow.c*, sua tela exibirá a seguinte saída:

```
10 elevado a -2 é 0.010000
10 elevado a -1 é 0.100000
10 elevado a 0 é 1.000000
10 elevado a 1 é 10.000000
10 elevado a 2 é 100.000000
C:\>
```

Nota: Muitos compiladores C também oferecem a função powl, que suporta valores do tipo long double. Da mesma forma, se você estiver trabalhando com valores complexos, o arquivo de cabeçalho complex.h definirá um protótipo de função para pow que trabalha com números complexos.

344 CALCULANDO O RESULTADO DE 10^X

Na dica anterior você viu como usar a função *pow* para determinar o resultado de um valor elevado a uma dada potência. Algumas vezes seus programas precisarão calcular o resultado de 10^x. Nesses casos, você poderá usar a função *pow*, ou, se seu compilador suportar (como o *Turbo C++ Lite*), poderá usar *pow10* de C, como mostrado aqui:

```
#include <math.h>

double pow(int potencia);
```

O programa a seguir, *pow10.c*, ilustra como usar a função *pow10*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("10 elevado a -1 é %f\n", pow10(-1));
   printf("10 elevado a 0 é %f\n", pow10(0));
   printf("10 elevado a 1 é %f\n", pow10(1));
   printf("10 elevado a 2 é %f\n", pow10(2));
 }
```

Quando você compilar e executar o programa *pow10.c*, sua tela exibirá a seguinte saída:

```
10 elevado a -1 é 0.100000
10 elevado a 0 é 1.000000
10 elevado a 1 é 10.000000
10 elevado a 2 é 100.000000
c:\>
```

Nota: Muitos compiladores C também fornecem a função pow10l, que suporta valores do tipo long double.

345 GERANDO UM NÚMERO ALEATÓRIO

Dependendo do seu programa, algumas vezes você precisa gerar um ou mais números aleatórios. Para esses casos, C oferece duas funções, *rand* e *random*, que retornam números inteiros aleatórios, como mostrados aqui:

```
#include <stdlib.h>

int rand(void);
int random(int teto);
```

A primeira função, *rand*, retorna um número inteiro aleatório no intervalo de 0 até *RAND_MAX* (definido em *stdlib.h*). A segunda função, *random*, retorna um número aleatório no intervalo até *teto*, onde *teto* é o tamanho do número aleatório máximo, que a função chamadora passa para a função *random*. O programa a seguir, *aleatori.c*, ilustra como usar ambos os geradores de números aleatórios:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   int i;

   printf("Valores de rand\n");
   for (i = 0; i < 100; i++)
     printf("%d ", rand());

   printf("Valores de random(100))\n");
   for (i = 0; i < 100; i++)
     printf("%d ", random(100));
 }
```

MAPEANDO VALORES ALEATÓRIOS PARA UM INTERVALO ESPECÍFICO 346

Na Dica 345, você aprendeu que as funções *rand* e *random* retornam números aleatórios. Quando seu programa gerar números aleatórios, algumas vezes ele precisará mapear os valores para um intervalo específico. Se você estiver trabalhando com valores inteiros, poderá usar a função *random* usando um parâmetro para especificar o valor mais alto no intervalo dos números aleatórios. No entanto, se você estiver trabalhando com valores em ponto flutuante, tais como valores no intervalo de 0.0 até 1.0, poderá dividir o número por uma constante para derivar um número em ponto flutuante. Para mapear uma série de inteiros aleatórios em uma série de números de ponto flutuante, simplesmente divida o número aleatório pelo limite superior do número aleatório, como mostrado aqui:

```
random(100)/100;
```

Esse exemplo gera um valor aleatório no intervalo de 0.01 até 0.99. Se seu programa precisar de mais dígitos no número aleatório em ponto flutuante, você poderá gerar um número aleatório até 1000 e dividir por 1000, como mostrado aqui:

```
random(1000)/1000;
```

O exemplo anterior gera um valor aleatório no intervalo de 0.001 até 0.999. Se seu programa precisar de mais precisão nos seus números aleatórios, simplesmente aumente o tamanho no inteiro aleatório máximo e a constante pela qual você divide o resultado de *random*. O programa a seguir, *mapa_ale.c*, mapeia os números aleatórios para o intervalo de 0.0 até 1.0 e valores inteiros para o intervalo de -5 até 5:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   int i;

   printf("Valores de random\n");
   for (i = 0; i < 10; i++)
```

```
        printf("%f\n", random(100)/100);
    printf("Valores de random(-5) até random(5)\n");
    for (i = 0; i < 100; i++)
        printf("%d\n", random(10)-5);
}
```

347 INICIALIZANDO O GERADOR DE NÚMEROS ALEATÓRIOS

A Dica 345 apresentou as funções *rand* e *random* de C que você usará dentro de seus programas para gerar números aleatórios. Quando você trabalhar com números aleatórios, algumas vezes desejará controlar as séries de números que o gerador de número aleatório cria (para que você possa testar o processamento do seu programa com o mesmo conjunto de números). Também haverá situações onde você irá querer que o gerador crie números reais aleatoriamente. O processo de atribuir o número inicial do gerador de números aleatórios é chamado de *inicializar o gerador*. Para lhe ajudar a inicializar os geradores de números aleatórios, C fornece duas funções, *randomize* e *srand*, como mostrado aqui:

```
#include <stdlib.h>

void randomize(void);
void srand(unsigned semente);
```

A primeira função, *randomize*, usa o relógio do PC para produzir uma semente aleatória. Por outro lado, a segunda função, *srand*, lhe permite especificar o valor inicial do gerador de números aleatórios. Seus programas podero usar *srand* para controlar o intervalo de números que o gerador de números aleatórios criará. O programa a seguir, *inicial.c*, ilustra como usar as funções *srand* e *randomize*:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

void main(void)
 {
   int i;

   srand(100);
   printf("Valores de rand\n");
   for (i = 0; i < 5; i++)
      printf("%d ", rand());
   printf("\nMesmos 5 números\n");
   srand(100);
   for (i = 0; i < 5; i++)
      printf("%d ", rand());
   randomize();
   printf("\n5 números diferentes\n");
   for (i = 0; i < 5; i++)
      printf("%d ", rand());
}
```

348 CALCULANDO A RAIZ QUADRADA DE UM VALOR

Quando seus programas calculam expressões matemáticas, eles freqüentemente precisam executar operações de raiz quadrada. Para ajudar seus programas a realizar as operações de raiz quadrada, C fornece a função *sqrt*, como mostrado aqui:

```
#include <math.h>

double sqrt(double valor);
```

A função *sqrt* trabalha somente com valores positivos Se seu programa chamar *sqrt* com um valor negativo, *sqrt* definirá a variável global *errno* como *EDOM*. O programa a seguir, *sqrt.c*, ilustra como usar a função *sqrt*:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   double valor;

   for (valor = 0.0; valor < 10.0; valor += 0.1)
      printf("Valor %f Raiz quadrada %f\n", valor,
           sqrt(valor));
 }
```

Nota: Muitos compiladores C também oferecem a função sqrtl, que retorna a raiz quadrada de um valor long double.

CRIANDO UMA ROTINA DE TRATAMENTO DE ERRO MATEMÁTICO 349

Várias funções apresentadas nesta seçao detectam erros de intervalo e de extravasamento. Por padrão, quando esses erros ocorrem, as funções chamam uma função especial chamada *matherr*, que efetua processamento adicional, tal como atribuir à variável global *errno* um número de erro específico Se seus programas definirem suas próprias funções *matherr*, as rotinas matemáticas de C chamarão sua rotina de tratamento personalizada. Quando as rotinas matemáticas chamarem sua função *matherr*, elas passarão para *matherr* um ponteiro para uma variável do tipo *exception*, como mostrado aqui:

```
struct  exception
 {
   int type;
   char *function;
   double arg1, arg2, retval;
 };
```

O membro type contém uma constante que descreve o tipo do erro. A Tabela 349 descreve os valores dos erros.

Tabela 349 Constantes de C que descrevem os erros matemáticos.

Valor do Erro	Significado
DOMAIN	Um argumento está fora do domínio de valores que a função suporta
OVERFLOW	Um argumento produz um resultado que extravasa o tipo do resultado
SING	Um argumento produz um resultado em uma singularidade
TLOSS	Um argumento produz um resultado no qual todos os dígitos da precisão são perdidos
UNDERFLOW	Um argumento produz um resultado que extravasa o tipo de resultado

O membro *function* contém o nome da rotina que experimentou o erro. Os membros *arg1* e *arg2* contêm os parâmetros que a função que está experimentando o erro passou para *matherr*; enquanto *retval* contém um valor de retorno padrão (que você pode atribuir). Se *matherr* não puder determinar a causa específica do erro, *matherr* exibirá uma mensagem de erro genérica na tela. O programa a seguir, *matherr.c*, ilustra como usar uma rotina de tratamento de erro personalizada:

```
#include <stdio.h>
#include <math.h>

void main(void)
 {
   printf("A raiz quadrada de -1 é %f\n", sqrt(-1.0));
 }

int matherr(struct exception *erro)
 {
```

```
  switch (erro->type)
  {
    case DOMAIN: printf("Erro de domínio\n");
                 break;
    case PLOSS: printf("Erro de perda de precisão parcial\n");
                 break;
    case OVERFLOW: printf("Erro de extravasamento \n");
                 break;
    case SING: printf("Erro na singularidade\n");
                 break;
    case TLOSS: printf("Erro de perda da precisão total\n");
                 break;
    case UNDERFLOW: printf("Erro de extravasamento\n");
                 break;
  };
  printf("O erro ocorreu em %s valores %f\n", erro->name, erro->arg1);
  erro->retval = 1;
  return(1);
 }
```

*Nota: A função **matherr** somente pega erros de domínio e de extravasamento. Para detectar erros de divisão por zero, use **signal**. Muitos compiladores C também suportam a função **matherrl**, que suportam argumentos do tipo **long double**.*

350 DETERMINANDO A UNIDADE DE DISCO ATUAL

Se seus programas trabalham no ambiente do DOS, algumas vezes você precisa determinar a unidade de disco atual. Nesses casos, seus programas podem usar a função *getdisk*, como mostrado aqui:

```
#include <dir.h>

int getdisk(void);
```

A função retorna um número da unidade de disco, onde 1 é a unidade A, 2 é a unidade B, e assim por diante. O programa a seguir, *LeUnid.c*, usa a função *getdisk* para exibir a letra da unidade de disco atual:

```
#include <stdio.h>
#include <dir.h>

void main(void)
 {
   printf('A unidade atual é %c\n", getdisk() + 'A');
 }
```

*Nota: O CD-ROM que acompanha este livro inclui o arquivo **win_getd.c**, que efetua a mesma tarefa que o programa LeUnid.c, mas trabalha somente sob o Windows 95 e sob o Windows NT.*

351 SELECIONANDO A UNIDADE ATUAL

Na dica anterior você aprendeu como usar a função *getdisk* para determinar a unidade de disco atual em um ambiente baseado no DOS. Exatamente como algumas vezes seus programas precisam determinar a unidade de disco atual, outras vezes eles precisam selecionar uma unidade de disco específica. Em tais casos, seus programas podem usar a função *setdisk*, como mostrado aqui:

```
#include <dir.h>

int setdisk(int unidade);
```

O parâmetro *unidade* é um valor inteiro que especifica a unidade desejada, onde 0 é a unidade A, 1 é a unidade B, e assim por diante. A função retorna o número de unidades de disco presentes no sistema. O programa a seguir, *selec_c.c*, usa a função *setdisk* para selecionar a unidade C como a unidade atual. O programa também exibe um contador do número de unidades disponíveis (como definido pela entrada *LASTDRIVE* no arquivo *config.sys*):

O parâmetro *unid* é um valor inteiro que especifica a unidade desejada, onde 0 é a unidade A, 1 é a unidade B, e assim por diante. A função retorna o número de unidades de disco presentes no sistema. O programa a seguir, *selec_c.c*, usa a função *setdisk* para selecionar a unidade C como a unidade atual. O programa também exibe um contador do número de unidades disponíveis (como definido pela entrada *LASTDRIVE* no arquivo *config.sys*).

```
#include <stdio.h>
#include <dir.h>

void main(void)
 {
   int conta_unid;
   conta_unid = setdisk(3);
   printf("O número de unidades disponíveis é %d\n", conta_unid);
 }
```

Nota: O CD-ROM que acompanha este livro inclui o arquivo ***win_setd.cpp****, que efetua a mesma tarefa que o programa select_c.c, mas trabalha apenas sob o Windows 95 e sob o Windows NT.*

DETERMINANDO O ESPAÇO DISPONÍVEL NO DISCO 352

Quando seus programas armazenam quantidades consideráveis de informações em um disco — não importando se esse disco é um disquete, um disco rígido, ou outro tipo — cada programa deve controlar o espaço disponível no disco para reduzir a possibilidade de ficar sem espaço durante uma operação crítica no disco. Se você estiver trabalhando em um sistema baseado no DOS, seus programas podem usar a função *getdfree*. A função *getdfree* retorna uma estrutura do tipo *dfree*, como mostrado aqui:

```
struct dfree
 {
   unsigned df_avail;    // Agrupamentos disponíveis
   unsigned df_total;    // Total de agrupamentos
   unsigned df_bsec;     // Bytes por setor
   unsigned df_sclus;    // Setores por agrupamento
 };
```

O formato da função *getdfree* é como segue:

```
#include <dos.h>

void getdfree(unsigned char unid, struct dfree *dtabela);
```

O parâmetro *unid* especifica a unidade desejada, onde 1 é a unidade A, 2 é a unidade B, e assim por diante. O programa a seguir, *discoliv.c*, usa a função *getdfree* para obter informações específicas sobre a unidade de disco atual:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct dfree info_disco;
   long espaco_disco;

   getdfree(3, &info_disco);
   espaco_disco = (long) info_disco.df_avail *
       (long) info_disco.df_bsec *
       (long) info_disco.df_sclus;

   printf("Espaço disponível no disco %ld\n", espaco_disco);
 }
```

Nota: O CD-ROM que acompanha este livro inclui o arquivo win_free.cpp, que efetua a mesma tarefa que o programa diskfree.c, mas trabalha apenas sob o Windows 95 e sob o Windows NT.

353 CUIDADO COM O DBLSPACE

Algumas dicas nesta seção mostram modos de executar operações de leitura e escrita absoluta no disco que trabalham com os setores de um disco. Antes que seus programas efetuem operações de E/S de baixo nível, certifique-se de que o disco que será lido não é um disco compactado com conteúdo que o *dblspace* ou outro utilitário de terceiros compactou anteriormente. Os discos compactados armazenam informações em setores seqüenciais. Se você gravar um setor de um disco compactado, correrá um risco considerável de corromper o disco — perdendo as informações que ele contém. Como regra, a maioria dos programa não precisa executar essas operações de leitura e gravação de baixo nível no disco. Se você estiver desenvolvendo um programa utilitário de disco, tal como o *undelete*, certifique-se de que você saberá como testar e trabalhar com discos compactados antes de iniciar.

354 LENDO AS INFORMAÇÕES DE UMA TABELA DE ALOCAÇÃO DE ARQUIVOS

Se você estiver trabalhando em um sistema baseado no DOS, a tabela de alocação de arquivos controlará quais partes do seu disco estarão em uso, quais partes estar danificadas e quais partes estarão disponíveis (para armazenamento de arquivos e do programa). Se seus programas efetuam operações de baixo nível no disco, algumas vezes você precisa conhecer informações, tais como o tipo do disco, bytes por setor, número de setores por agrupamento e o número de agrupamentos no disco. Em tais casos, seus programas podem usar as funções *getfat* ou *getfatd*, como mostrado aqui:

```
#include <dos.h>

void getfat(unsigned char unid, struct fatinfo *fat);
void getfatd(struct fatinfo *fat);
```

A função *getfat* lhe permite especificar a unidade desejada, enquanto que *getfatd* retorna as informações para a unidade atual. Para especificar uma letra de unidade de disco para a função *getfat*, especifique um valor onde 1 é a unidade A, 2 é a unidade B, 3 é a unidade C, e assim por diante. As funções *getfat* e *getfatd* atribuem as informações a uma estrutura do tipo *fatinfo*, como mostrado aqui:

```
struct fatinfo
 {
   char fi_sclus;        // Setores por agrupamento (cluster)
   char fi_fatid;        // Tipo do disco
   unsigned fi_nclus;    // Agrupamentos por disco
   int fi_bysec;         // Bytes por setor
 };
```

O programa a seguir, *getfatd.c*, usa a função *getfatd* para exibir informações sobre a unidade de disco atual:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct fatinfo fat;

   getfatd(&fat);

   printf("Setores por agrupamento %d\n", fat.fi_sclus);
   printf("Agrupamentos por disco %u\n", fat.fi_nclus);
   printf("Bytes por agrupamento %d\n", fat.fi_bysec);
   printf("Tipo do disco %x\n", fat.fi_fatid & 0xFF);
 }
```

Nota: Se seu computador está rodando o Windows NT e você particionou a unidade como uma unidade NT File System (NTFS), não há uma tabela FAT para você acessar. Para aprender mais sobre NTFS, visite o site do Departamento de Ciência da Computação da Universidade de Yale em http://pclt.cis.yale.edu/pclt/BOOT/IFS.HTM.

COMPREENDENDO A IDENTIFICAÇÃO DO DISCO 355

Na Dica 354 você usou as funções *getfat* e *getfatd* para obter informações sobre a unidade de disco atual. Como viu, essas funções retornam um byte chamado *fi_fatid*, que contém uma representação da identificação do disco do DOS. A Tabela 355 especifica os possíveis valores para *fi_fatid*.

Tabela 355 Identificações dos valores que o DOS retorna.

Valor (Hexa)	Tipo do Disco
F0H	1.44Mb ou 2.88Mb de 3,5 polegadas
	Disco Zip
F8H	Disco rígido
	Unidade de CD-ROM
F9H	720Kb de 3.5 ou 1.2Mb de 5.25 polegadas
FAH	320Kb de 5.25 polegadas
FCH	180Kb de 5 25 polegadas
FDH	360Kb de 5.25 polegadas
FEH	160Kb de 5.25 polegadas
FFH	320Kb de 5.25 polegadas

Nota: O CD-ROM que acompanha este livro inclui o arquivo ***win_did.cpp****, que lista as identificações de disco sob o Windows 95 ou o Windows NT e mostra-as na tela.*

EFETUANDO UMA LEITURA OU GRAVAÇÃO ABSOLUTA DE SETOR 356

Se você trabalha no ambiente do DOS, o DOS lhe permite efetuar operações absolutas de leitura e gravação no nível do setor. Normalmente, seus programas usam os serviços do DOS para efetuar essas operações. No entanto, para tornar essas operações mais fáceis de executar, muitos compiladores C oferecem as funções *absread* e *abswrite*, como mostrado aqui:

```
#include <dos.h>

int absread(int unidade, int num_setores, long setor_inicial, void *buffer);
int abswrite(int unidade, int num_setores, long setor_inicial, void *buffer);
```

O parâmetro *unidade* especifica a unidade de disco que você quer ler, onde 0 é a unidade A, 1 é a unidade B, e assim por diante. O parâmetro *num_setores* especifica o número de setores que você quer ler ou gravar, iniciando no setor que o parâmetro *setor_inicial* especifica. Finalmente, o parâmetro *buffer* é um ponteiro para o buffer no qual as informações são lidas ou a partir das quais a saída é gravada. Se as funções são bem-sucedidas, elas retornam o valor 0. Se um erro ocorre, as funções retornam o valor -1. O programa a seguir, *chc_disc.c*, lê todo setor no disco C. Se o programa encontrar erros ao ler um setor, ele exibirá o número do setor:

```
#include <stdio.h>
#include <dos.h>
#include <alloc.h>

void main(void)
 {
   struct fatinfo fat;
   long sector, total_setores;
   void *buffer;

   getfat(3, &fat);
   total_setores = fat.fi_nclus * fat.fi_sclus;
   if ((buffer = malloc(fat.fi_bysec)) == NULL)
      printf("Erro ao alocar o buffer do setor\n");
   else
```

```
    for (sector = 0; sector < total_setores; sector++)
      if (absread(2, 1, sector, buffer) == -1)
       {
         printf("\n\007Erro no setor %ld pressione Enter\n", sector);
         getchar();
       }
      else
        printf("Lendo o setor %ld\r", sector);
}
```

Nota: Embora você possa efetuar leituras e gravações absolutas de setor no Windows, o modo como o Windows grava informações no disco torna as leituras e gravações absolutas perigosas e inconsistentes. Você deve evitar as atividades absolutas no disco no Windows e processar as leituras e escritas por meio da Interface de Programação de Aplicativos (Application Programming Interface — API) do Windows.

357 EFETUANDO E/S NO DISCO BASEADA NO BIOS

Quando seus programas efetuam operações em arquivo, usam os serviços do DOS para manipular os arquivos. Os serviços do DOS, por sua vez, chamam outros serviços do DOS para ler e gravar setores lógicos no disco. Para efetuar as operações reais de E/S no disco, os serviços do DOS então chamam os serviços do disco da BIOS. Em tais casos, seus programas podem usar a função *biosdisk*, como mostrado aqui:

```
#include <bios.h>

int biosdisk(int operação, int unidade, int cabeça, int trilha, int setor,
int conta_setor, void *buffer);
```

O parâmetro *unidade* especifica o número da unidade, onde 0 é a unidade A, 1 é a unidade B, e assim por diante. Para um disco rígido, 0x80 é o primeiro disco rígido, 0x81 é a segunda unidade, e assim sucessivamente. Os parâmetros *cabeça*, *trilha*, *setor* e *conta_setor* especificam os setores do disco físico que você quer ler ou gravar. O parâmetro *buffer* é um ponteiro para o buffer no qual *biosdisk* lê os dados ou a partir do qual *biosdisk* grava os dados. Finalmente, o parâmetro *operação* especifica a função desejada. A Tabela 357.1 lista as operações válidas.

Tabela 357.1 Operações válidas de *biosdisk*.

Operação	Função
0	Reinicializa o sistema do disco
1	Retorna o status da última operação no disco
2	Lê o número especificado de setores
3	Grava o número especificado de setores
4	Verifica o número especificado de setores
5	Formata a trilha especificada — o buffer contém uma tabela de localizações ruins
6	Formata a trilha especificada, marcando os setores ruins
7	Formata a unidade começando na trilha especificada
8	Retorna os parâmetros da unidade nos primeiros quatro bytes do buffer
9	Inicializa a unidade
10	Efetua uma leitura longa — 512 bytes de setor mais 4 extras
11	Efetua uma escrita longa — 512 bytes de setor mais 4 extras
12	Efetua um posicionamento (seek) no disco
13	Reinicialização alternativa do disco
14	Lê buffer de setor
15	Grava buffer de setor
16	Testa unidade pronta
17	Recalibra a unidade
18	Efetua o diagnóstico da RAM da controladora
19	Efetua o diagnóstico da unidade
20	Executa o diagnóstico interno da controladora

Se bem-sucedidas, as funções retornam o valor 0. Se ocorrer um erro, o valor de retorno da função especifica o erro. A Tabela 357.2 lista os valores de erro.

Tabela 357.2 Valores de status de erro retornados por *biosdisk*.

Valor do Erro	Erro
0	Bem-sucedida
1	Comando inválido
2	Marca de endereço não-encontrada
3	Disco protegido contra gravação
4	Setor não-encontrado
5	A reinicialização do disco rígido falhou
6	Troca do disco
7	Falha no parâmetro atividade da unidade
8	Erro de DMA
9	DMA fora do limite de 64Kb
10	Setor ruim
11	Trilha ruim
12	Trilha não-suportada
16	Erro de leitura CRC/ECC
17	Dados corrigidos CRC/ECC
32	Falha na controladora
64	Falha na operação de posicionamento (seek)
128	Sem resposta
170	Disco rígido não-preparado
187	Erro não-definido
204	Falha na gravação
224	Erro de status
255	A operação Sense falhou

Nota: Muitos compiladores também oferecem uma função chamada ***_bios_disk****, que efetua processamento idêntico a* ***biosdisk****, com a exceção de que seus programas passam para a função uma estrutura do tipo* ***diskinfo_t****, que contém os valores drive (unidade), head (cabeça), track (trilha), sector (setor) e sector_count (conta_setor).*

Nota: Embora você possa usar bios_disk para executar E/S no disco baseada na BIOS, os métodos que o Windows usa para gravar informações no disco tornam a E/S no disco baseada na BIOS perigosa e inconsistente. Evite a E/S no disco baseada na BIOS no Windows e processe suas leituras e escritas por meio da Interface de Programação de Aplicativos (API) do Windows.

TESTANDO A PRONTIDÃO DE UMA UNIDADE DE DISQUETE 358

Na dica anterior você aprendeu como usar a função *biosdisk* para chamar os serviços de disco da BIOS. Uma operação útil que a função *biosdisk* pode executar é testar se uma unidade de disquete contém um disco e está pronta para ser acessada. O programa a seguir, *testa_a.c*, usa a função *biosdisk* para verificar a unidade de disquete:

```
#include <stdio.h>
#include <bios.h>

void main(void)
 {
   char buffer[8192];

   // Tenta ler a cabeça 1, trilha 1, setor 1
   if (biosdisk(2, 0, 1, 1, 1, 1, buffer))
     printf("Erro ao acessar a unidade\n");
   else
     printf("Unidade pronta\n");
 }
```

Nota: O CD-ROM que acompanha este livro inclui o arquivo ***win_a.cpp****, que efetua a mesma tarefa que o programa testa_a.c, mas trabalha apenas sob o Windows 95 ou sob o Windows NT.*

359 ABRINDO UM ARQUIVO USANDO FOPEN

Muitos programas C que você criar, irão armazenar e ler informações de um arquivo. Antes que seus programas possam ler ou gravar informações em um arquivo, você precisará abrir o arquivo. A função *fopen* permite que seus programas abram um arquivo. O formato de *fopen* é como segue:

```
#include <stdio.h>

FILE *fopen(const char *nomearq, const char *modo);
```

O parâmetro *nomearq* é uma string de caracteres que contém o nome do arquivo desejado, tal como "*c:\arqdados.dat*". O parâmetro *modo* especifica como você quer usar o arquivo — para ler, gravar ou anexar. A Tabela 359 descreve o valores de modo que *fopen* suporta.

Tabela 359 Os valores de modo suportados por *fopen*.

Modo	Significado
a	Abre um arquivo para operações de anexação — se o arquivo não existir, o sistema operacional o criará
r	Abre um arquivo existente para operações de leitura
w	Abre um novo arquivo para saída — se um arquivo com o mesmo nome existir, o sistema operacional irá sobrescrever o arquivo
r+	Abre um arquivo existente para leitura e gravação
w+	Abre um novo arquivo para leitura e gravação — se um arquivo com o mesmo nome existir, o sistema operacional irá sobrescrever o arquivo
a+	Abre um arquivo para operações de anexação e leitura — se o arquivo não existir, o sistema operacional criará o arquivo

A função *fopen* retorna um ponteiro (chamado *ponteiro de arquivo*) para uma estrutura do tipo *FILE* que o arquivo de cabeçalho *stdio.h* define. Seus programas usarão o ponteiro de arquivo para suas operações de entrada e saída. Se a função *fopen* não puder abrir o arquivo especificado, ela retornará o valor *NULL*. Seus programas sempre devem testar o valor de retorno de *fopen* para garantir que abriu o arquivo com sucesso, como mostrado aqui:

```
if((pa = fopen("NOMEARQ.EXT", "r")) != NULL)
  {
    // Arquivo aberto com sucesso
  }
else
  {
    // Erro ao abrir o arquivo
  }
```

Dentro do seu programa, você precisa declarar a variável ponteiro de arquivo como segue:

```
void main(void)
 {
   FILE *pa   // Ponteiro para uma estrutura do tipo FILE
```

Muitos programas abrem um arquivo para entrada e outro para saída. Em tais casos, você poderia declarar dois ponteiros de arquivo, como mostrado aqui:

```
FILE *entrada, *saida;
```

Muitas dicas nesta seção usam *fopen* para abrir um arquivo para as operações de leitura, gravação ou anexação.

COMPREENDENDO A ESTRUTURA FILE 360

Como você aprendeu, quando seus programas efetuam operações de entrada e saída, elas normalmente declaram ponteiros de arquivo usando a estrutura *FILE*, como mostrado aqui:

```
FILE *entrada, *saida;
```

Se você examinar o arquivo de cabeçalho *stdio.h*, encontrará a definição da estrutura *FILE*. No caso do *Turbo C++ Lite*, a estrutura tem a seguinte forma:

```
typedef  struct
 {
   short level;              // Nível do buffer cheio/vazio
   unsigned flags;           // Sinalizadores de status
   char fd;                  // Descritor de arquivo
   unsigned char hold;       // Caractere ungetc se não existir um buffer
   short bsize;              // Tamanho do buffer
   unsigned char *buffer;    // Buffer de transferência
   unsigned char *curp;      // Ponteiro ativo atual
   unsigned istemp;          // Indicador de arquivo temporário
   short token;              // Usado para verificação da validade
 } FILE;                     // Esse é o objeto FILE
```

A estrutura *FILE* contém o *descritor de arquivo* de baixo nível que o sistema operacional usa para acessar o arquivo, o tamanho do buffer do arquivo e a localização, o buffer de caracteres que *unget* usa, um sinalizador que indica se o arquivo é um arquivo temporário, e outras variáveis sinalizadoras. Além disso, a estrutura *FILE* armazena o ponteiro de arquivo que controla sua localização atual dentro do arquivo.

Se você estiver trabalhando no ambiente do DOS, a maioria dos compiladores definirá uma matriz de tamanho fixo (normalmente 20) dos ponteiros de arquivo que contêm as informações para cada arquivo que seu programa abrirá. Se seu programa precisar abrir mais de 20 arquivos, consulte a documentação do seu compilador para conhecer os passos que precisará seguir para modificar o tamanho da matriz de ponteiros de arquivo.

FECHANDO UM ARQUIVO ABERTO 361

Exatamente como seus programas precisam abrir um arquivo antes de usá-lo, também devem fechar o arquivo quando não precisarem mais dele. Fechar um arquivo instrui o sistema operacional a esvaziar todos os buffers de disco associados com o arquivo e a liberar os recursos do sistema que o arquivo consumiu, tais como os dados de ponteiro de arquivo. A função C *fclose* fecha o arquivo associado com o ponteiro de arquivo especificado, como mostrado aqui:

```
#include <stdio.h>

int fclose(FILE *pont_arquivo);
```

Se *fclose* for bem-sucedida, retornará o valor 0. Se ocorrer um erro, *fclose* retornará a constante *EOF*, como mostrado aqui:

```
if (fclose(pa) == EOF)
    printf("Erro ao fechar o arquivo de dados\n");
```

À medida que você for examinando programas C, verá que a maioria deles não testa o valor de retorno de *fopen*, como mostrado aqui:

```
fclose(pa);
```

Na maioria dos casos, se uma operação fechar arquivo apresentar erro, o programa poderá fazer muito pouco para corrigir a situação. No entanto, se você estiver trabalhando com arquivos de dados críticos, deverá exibir uma mensagem de erro para o usuário para que ele possa examinar o conteúdo do arquivo.

Nota: Se você não chamar a função fclose, C fechará os arquivos abertos quando o programa terminar.

362 LENDO E GRAVANDO INFORMAÇÕES NO ARQUIVO UM CARACTERE DE CADA VEZ

Quando seus programas efetuam operações de entrada e saída em arquivos, podem ler e gravar dados um caractere de cada vez ou uma linha de cada vez. Para as operações de entrada e saída de caractere, seus programas podem usar as funções *fgetc* e *fputc*, cujos formatos são mostrados aqui:

```
#include <stdio.h>

int fgetc(FILE *pont_entrada);
int fputc(int caractere, FILE *pont_saida);
```

A função *fgetc* lê o caractere atual do arquivo de entrada especificado. Se o ponteiro de arquivo tiver chegado ao final do arquivo, *fgetc* retornará a constante *EOF*. A função *fputc* gravará um caractere na posição do ponteiro de arquivo atual dentro do arquivo de saída especificado. Se um erro ocorrer, *fputc* retornará a constante *EOF*. O programa a seguir, *copconf.c*, usa *fgetc* e *fputc* para copiar o conteúdo do arquivo do diretório-raiz *config.sys* para um arquivo chamado *config.tst*:

```
#include <stdio.h>

void main(void)
 {
   FILE *entrada, *saida;
   int letra;

   if ((entrada = fopen("\\CONFIG.SYS", "r")) == NULL)
     printf("Erro ao abrir \\CONFIG.SYS\n");
   else if ((saida = fopen("\\CONFIG.TST", "w")) == NULL)
     printf("Erro ao abrir \\CONFIG.TST\n");
   else
     {
       // Lê e grava cada caractere no arquivo
       while ((letra = fgetc(entrada)) != EOF)
          fputc(letra, saida);
       fclose(entrada);      // Fecha o arquivo entrada
       fclose(saida);        // Fecha o arquivo saida
     }
 }
```

363 COMPREENDENDO O PONTEIRO DE POSIÇÃO DO PONTEIRO DE ARQUIVO

A Dica 360 apresentou a estrutura *FILE*. Como você aprendeu, um dos campos da estrutura armazena um *ponteiro da posição* para a localização atual dentro do arquivo. Quando você abre um arquivo para operações de leitura ou gravação, o sistema operacional define o ponteiro da posição no início do arquivo. Toda vez que você ler ou gravar um caractere, o ponteiro da posição avançará um caractere. Se você ler uma linha de texto do arquivo, o ponteiro da posição avançará para o início da próxima linha. Usando o ponteiro da posição, as funções de entrada e saída de arquivo sempre podem controlar a localização atual dentro do arquivo. Quando você abre um arquivo no modo de anexação, o sistema operacional define o ponteiro da posição no final do arquivo. Em dicas posteriores você aprenderá a alterar o ponteiro da posição para posições específicas no arquivo usando as funções *fseek* e *fsetpos*. A Tabela 363 especifica a localização na qual *fopen* coloca o ponteiro da posição quando você abre o arquivo nos modos de leitura, gravação e anexação.

Tabela 363 Posicionamentos do ponteiro de posição no arquivo que resultam de uma chamada a *fopen*.

Modo de Abertura	Posição do Ponteiro do Arquivo
a	Imediatamente após o último caractere no arquivo
r	No início do arquivo
w	No final do arquivo

DETERMINANDO A POSIÇÃO ATUAL NO ARQUIVO 364

Na dica anterior você aprendeu como C controla a posição atual em arquivos abertos para as operações de entrada e saída. Dependendo do seu programa, algumas vezes você precisará determinar o valor do ponteiro de posição. Em tais casos, seus programas podem usar a função *ftell*, como mostrado aqui:

```
#include <stdio.h>

long int ftell(FILE *pont_arquivo);
```

A função *ftell* retorna um valor inteiro longo que especifica o byte de deslocamento a partir da posição atual no arquivo especificado. O programa a seguir, *exib_pos.c*, usa *ftell* para exibir informações do ponteiro de posição. O programa começa abrindo o arquivo do diretório-raiz *config.sys* no modo de leitura. O programa então usa *ftell* para exibir a posição atual. Em seguida, o programa lê e exibe o conteúdo do arquivo. Após encontrar o final do arquivo, o programa novamente usará *ftell* para exibir a posição atual, como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   FILE *entrada;
   int letra;

   if ((entrada = fopen("\\CONFIG.SYS", "r")) == NULL)
      printf("Erro ao abrir \\CONFIG.SYS\n");
   else
     {
       printf("A posição atual é o byte %d\n\n", ftell(entrada));
          // Lê e grava cada caractere no arquivo
       while ((letra = fgetc(entrada)) != EOF)
          fputc(letra, stdout);
       printf("\nA posição atual é o byte %d\n", ftell(entrada));
       fclose(entrada);      // Fecha o arquivo entrada
     }
 }
```

COMPREENDENDO OS CANAIS DE ARQUIVOS 365

Muitos livros e revistas referem-se aos ponteiros de arquivos de C como *ponteiros streams (ou de canais) de arquivos*. Ao contrário de muitas outras linguagens de programação, C não assume que os arquivos contêm informações em um formato específico. Em vez disso, C considera todos os arquivos como nada mais que uma coleção de bytes. À medida que você lê um arquivo, lê um byte após o outro; em outras palavras, uma seqüência de bytes. Seus programas e funções, tais como *fgets*, precisam interpretar os bytes. Por exemplo, *fgets* considera o caractere de alimentação de linha como o final de uma linha e o início de outra. A função *fgets* faz essa interpretação de caracteres sozinha. Isto é, C não interpreta os bytes. À medida que você for escrevendo programas e funções que manipulam arquivos, pense nos arquivos como nada mais que uma coleção de bytes.

COMPREENDENDO AS TRADUÇÕES DE ARQUIVOS 366

As funções de manipulação de arquivos de C, tais como *fgets* e *fputs*, podem interpretar arquivos em uma de duas formas: modo *texto* e modo *binário*. Por padrão, as funções *fgets* e *fputs* usam o modo texto. No modo texto, funções, tais como *fputs*, que gravam informações em um arquivo, convertem o caractere de alimentação de linha para uma combinação retorno do carro e alimentação de linha. Durante uma operação de entrada, funções, tais

como *fgets*, convertem a combinação retorno do carro e alimentação de linha para um único caractere de alimentação de linha. Por outro lado, no modo binário, as funções não efetuam essas traduções de caractere. Para lhe ajudar a determinar o modo de tradução atual, muitos compiladores baseados no DOS e no Windows oferecem a variável global *_fmode*, que contém um dos valores listados na Tabela 366.

Tabela 366 Os valores constantes para *_fmode*.

Constante	Descrição
O_TEXT	Traduções no modo texto
O_BINARY	Traduções no modo binário

Por padrão, o valor de *fmode*, tanto sob o DOS quanto sob o Windows, é *O_TEXT*. O programa a seguir, *fmode.c*, exibe o valor atual da variável *_fmode*:

```
#include <stdio.h>
#include <fcntl.h>     //Contém a declaração de _fmode

void main(void)
 {
   if (_fmode == O_TEXT)
      printf("Traduções do modo texto\n");
   else
      printf("Traduções do modo binário\n");
 }
```

367 COMPREENDENDO A LINHA FILES=VALOR NO CONFIG.SYS

Se você estiver trabalhando em um ambiente baseado no DOS, a linha FILES no arquivo *config.sys* especificará o número de arquivos que o sistema pode abrir ao mesmo tempo (em um ambiente baseado no Windows, o Windows limita o número de arquivos abertos com base na memória disponível do sistema, espaço no disco, uso de outros recursos etc.). Como discutido rapidamente nesta seção, Arquivos, Diretórios e Discos, o DOS usa os primeiros cinco indicativos (*handles*) de arquivos para *stdin*, *stdout*, *stderr*, *stdaux* e *stdprn*. Por padrão, o DOS oferece suporte para oito indicativos de arquivo. Como esse número é pequeno para quase todos os programas, exceto os mais simples, a maioria dos usuários aumenta o número de indicativos de arquivo para 20 ou 30, como mostrado aqui:

```
FILES=30
```

A linha FILES define o número de arquivos que o DOS pode abrir — não o número que cada programa rodando sob o DOS pode abrir. Por exemplo, se você estiver rodando programas residentes na memória, eles poderão abrir arquivos sem que você saiba. Se você definir a linha FILES como um número grande de indicativos (o DOS permite até 255), isso não significa que seus programas C poderão abrir todos esses arquivos. Existem dois problemas com a abertura de um grande número de arquivos dentro de seus programas C. Primeiro, a maioria dos compiladores restringe o tamanho da matriz de ponteiros de arquivos para 20. Antes de poder abrir mais do que 20 arquivos, você precisa alterar o tamanho da matriz. Segundo, como você aprenderá, o DOS restringe em 20 o número de arquivos que um ponteiro pode abrir. Antes de poder abrir mais de 20 arquivos, você precisa usar um serviço do DOS que pede que ele suporte mais do que 20 arquivos abertos para o programa atual.

Nota: A Dica 369 explicará os indicativos de arquivo.

368 USANDO E/S EM ARQUIVOS DE BAIXO NÍVEL E DE ALTO NÍVEL

Quando seus programas C trabalham com arquivos, podem efetuar dois tipos de operações de entrada e de saída: *baixo nível* e *alto nível*. Todas as dicas apresentadas até este ponto usaram as capacidades de alto nível (ou baseadas em canais (streams), tais como *fopen*, *fgets* e *fputs*. Quando você usa as funções de E/S de arquivo de alto nível, elas, por sua vez, usam serviços do sistema operacional que estão baseados em *indicativos de arquivo*. A bi-

blioteca de execução de C fornece funções de baixo nível que seus programas podem usar. Em vez de trabalharem com um ponteiro *stream*, as funções de baixo nível usam *descritores de arquivo*. A Tabela 368 descreve resumidamente várias das funções de baixo nível mais comumente usadas.

Tabela 368 Funções comuns de arquivo de baixo nível.

Nome da Função	Propósito
close	Fecha o arquivo associado com o indicativo de arquivo específico esvaziando os buffers de arquivo
creat	Cria um arquivo para as operações de saída, retornando um indicativo de arquivo
open	Abre um arquivo existente para entrada ou saída, retornando um indicativo de arquivo
read	Lê um número específico de bytes a partir do arquivo associado com um determinado indicativo de arquivo
write	Grava um número específico de bytes no arquivo associado com um determinado indicativo

Quando você escrever seus programas, sua escolha de usar funções de alto nível ou de baixo nível dependerá de sua preferência pessoal. No entanto, tenha em mente que a maioria dos programadores tem uma melhor compreensão das funções de manipulação de arquivo de alto nível de C. Como resultado, se você usar as funções de alto nível, tais como *fopen* e *fgets*, mais programadores irão prontamente compreender o código do seu programa.

Nota: A seção E/S no Windows, mais à frente, discute em detalhes a E/S em arquivo de baixo nível e de alto nível sob o Windows.

COMPREENDENDO OS INDICATIVOS DE ARQUIVOS 369

Como você sabe, a linha FILES no arquivo *config.sys* lhe permite especificar o número de indicativos de arquivo que o DOS suporta. Basicamente, um indicativo de arquivo é um valor inteiro que define de forma inequívoca um arquivo aberto. Ao usar as funções de E/S em arquivo de baixo nível, você declarará os indicativos de arquivo do seu programa como tipo *int*, como mostrado aqui:

```
int indic_entrada, indic_saida;
```

As funções *open* e *creat* retornarão descritores de arquivo ou o valor -1 se a função não puder abrir o arquivo:

```
int arq_novo, arq_antigo;
arq_novo = creat("NOMEARQ.NOV", S_IWRITE); // Cria um novo arquivo para saída
arq_antigo = open("NOMEARQ.ANT", O_RDONLY); // Abre um arquivo existente
para leitura
```

O DOS atribui a cada arquivo que você abrir ou criar um indicativo de arquivo distinto. O valor do indicativo é na verdade um índice na tabela de arquivos do processo, dentro da qual o DOS controla os arquivos abertos do programa.

COMPREENDENDO A TABELA DE ARQUIVOS DO PROCESSO 370

Quando você roda um programa no ambiente do DOS, o DOS controla os arquivos abertos do programa usando uma *tabela de arquivos do processo*. Dentro do prefixo do segmento do programa, o DOS armazena um ponteiro *far* para uma tabela que descreve os arquivos abertos do programa. Na verdade, a tabela contém entradas para uma segunda tabela, a *tabela de arquivos do sistema*, dentro da qual o DOS controla todos os arquivos abertos. A Figura 370 ilustra o relacionamento entre o indicativo de arquivo, a tabela de arquivos do processo e a tabela de arquivos do sistema.

Sob o Windows, o Gerenciador de Tarefas mantém a lista de todos os processos abertos, e o Windows usa a tabela de arquivos do sistema do DOS para manter a lista de todos os arquivos abertos. O CD-ROM que acompanha este livro inclui o programa *Task_Man.cpp*, que lista todos os programas atualmente abertos no sistema.

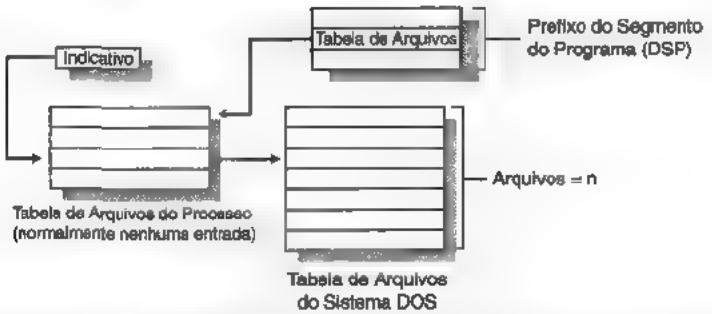

Figura 370 O relacionamento entre um indicativo de arquivo, a tabela de arquivos do processo e a tabela de arquivos do sistema.

371 VISUALIZANDO AS ENTRADAS DA TABELA DE ARQUIVOS DO PROCESSO

Como a Dica 370 descreveu, o DOS controla os arquivos abertos de um programa usando uma tabela de arquivos do processo. No deslocamento 18H, dentro do prefixo de segmento do programa, há uma matriz de valores inteiros. Os valores que compõem essa matriz especificam índices na tabela de arquivos do sistema DOS. Se um valor não está em uso, o sistema operacional define-o como FFH (decimal 255). O programa a seguir, *tabelarq.c*, exibirá os valores na tabela de arquivos do processo. Lembre-se, a tabela contém valores inteiros que servem como índices na tabela de arquivos do sistema:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

void main(void)
 {
   struct fcbs
    {
       char unidade;
       char nomearq[8];
       char extensao[3];
       int bloco_atual;
       int tamanho_registro;
    };
   typedef struct fcbs fcb;
   struct prefixo_segmento_programa
    {
       char near *int20;
       char near *segmento_prox_paragrafo;
       char reservado_1;
       char dos_dispatcher[5];
       char far *vetor_termino;
       char far *vetor_ctrlc;
       char far *critico_erro_vetor;
       char near *psp_pai;
       unsigned char tabela_arq[20];
       char near *segmento_bloco_ambiente;
       char far *armazenagem_pilha;
       int indicativos_disponiveis;
       char far *endereco_tabela_arq;
       char far *compartilha_psp_anterior;
       char reservado_2[20];
       char dos_int21_retf[3];
       char reservado_3[9];
       fcb fcb1;
```

```
        fcb fcb2;
        char reservado_4[4];
        char cauda_comando[128];
    } far *psp;
    int i;

    psp = (struct prefixo_segmento_programa far *) ((long) _psp << 16);
    for (i = 0; i < 20; i++)
        printf("Entrada %d contém %x\n", i, psp->tabela_arquivo[i]);
}
```

Quando você compilar e executar o programa *tabelarq.c*, verá que as primeiras cinco entradas na tabela de arquivos do processo estarão em uso. Essas entradas correspondem a *stdin, stdout, stderr, stdaux* e *stdprn*. Edite este programa e abra um ou mais arquivos antes de exibir as entradas da tabela de arquivos, e você encontrará mais entradas dentro da tabela de arquivos do processo.

COMPREENDENDO A TABELA DE ARQUIVOS DO SISTEMA 372

Os indicativos de arquivo são valores de índice na tabela de arquivos do processo, que, por sua vez, apontam para a tabela de arquivos do sistema. A tabela de arquivos do sistema armazena informações sobre todo arquivo que o DOS, um controlador de dispositivo, um programa residente em memória, ou seu programa abriu. A Figura 372 ilustra o conteúdo da tabela do sistema de arquivos.

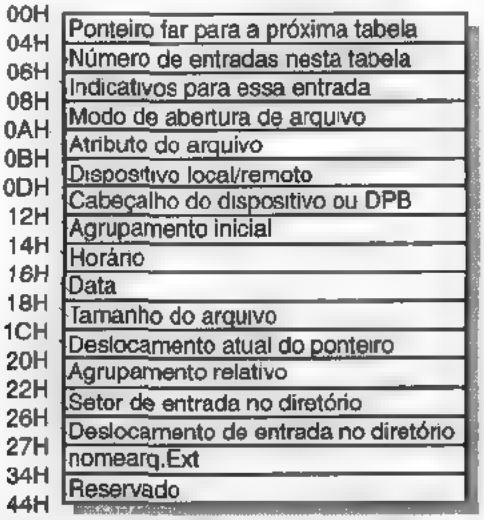

Figura 372 Conteúdo da tabela do sistema de arquivos do DOS.

O DOS, na verdade, divide a tabela do sistema em duas seções. A primeira seção contém cinco entradas. A segunda seção fornece espaço suficiente para o número de entradas que a linha FILES especifica no *config.sys* (menos 5 — as entradas que residem na primeira seção da tabela).

EXIBINDO A TABELA DO SISTEMA DE ARQUIVOS 373

O DOS armazena informações sobre todo arquivo aberto dentro da tabela de arquivos do sistema. Usando a lista de listas do DOS, discutida na seção DOS e BIOS, mais à frente, o programa a seguir, *tabsis.c*, exibe as entradas na tabela do sistema de arquivos:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

void main(void)
```

```
{
  union REGS regs_ent, regs_saida;
  struct SREGS segs;
  int i, j;
  int tam_estrutura;
  struct SystemTableEntry
   {
     struct SystemTableEntry far *proximo; // Próxima entrada SFT
     unsigned conta_arq;                          // Arquivos na tabela
     unsigned conta_indic; // Indicativos para este arquivo
     unsigned modo_abertura; // Modo de abertura do arquivo
     char atributo_arq;           // Byte de atributo
     unsigned local_remoto; // O bit 15 ligado indica remoto
     unsigned far *DPD; // Bloco de parâmetros da unidade
     unsigned agrupamento_inicial;
     unsigned hora;
     unsigned data;
     long tam_arquivo;
     long desloc_atual;
     unsigned agrupamento_relativo;
     long num_setor_diretorio;
     char desloc_entrada_diretorio;
     char nomearq_ext[11]; // Sem o ponto, espaço preenchido
                           // Ignora os campos SHARE, por exemplo
  } far *tabela_ptr, far *arquivo;
  long far *tabela_sistema;

    // Lê a versão do DOS
  regs_ent.x.ax = 0x3001;
  intdos (&regs ent, &regs_saida);
  if (regs_saida.h.al < 3)
   {
     printf("Este programa requer o DOS versão 3 ou posterior\n");
     exit (1);
   }
  else if (regs_saida.h.al == 3)
     tam_estrutura = 0x35;
  else if (regs_saida.h.al >= 4)
     tam_estrutura = 0x3B;
    // Lê a lista de ponteiros
  regs_ent.h.ah = 0x52;
  intdosx (&regs_ent, &regs_saida, &segs);
    // O ponteiro para a tabela de arquivos do sistema está no deslocamento 4
  tabela_sistema = MK_FP(segs.es, regs_saida.x.bx + 4);
  tabela_ptr = (struct SystemTableEntry far *) *tabela_sistema;
  do {
     printf("%d entradas na tabela\n", tabela_ptr->conta_arq);
     for (i = 0; i < tabela_ptr->conta_arq; i++)
      {
        arquivo = MK_FP(FP_SEG(tabela_ptr), FP_OFF(tabela_ptr) +
            (i * tam_estrutura));
        if (arquivo->conta_indic)
         {
           for (j = 0; j < 8; j++)
              if (arquivo->nomearq_ext[j] != ' ')
                 putchar(arquivo->nomearq_ext[j]);
              else
                 break;
           if (arquivo->nomearq_ext[8] != ' ')
                 putchar('.');
           for (j = 8; j < 11; j++)
```

```
                if (arquivo->nomearq_ext[j] != ' ')
                   putchar(arquivo->nomearq_ext[j]);
                printf(" %ld bytes %x atributo %d referências\n",
                   arquivo->tam_arquivo, arquivo->atributo_arq,
                   arquivo->conta_indic);
            }
         }
         tabela_ptr = tabela_ptr->proximo;
      } while (FP_OFF(tabela_ptr) != 0xFFFF);
   }
```

Quando você rodar o programa *tabsis.c* a partir do prompt do DOS, provavalmente a saída dele não será muito interessante. No entanto, se você tiver o Windows, use o ícone Prompt do MS-DOS para abrir uma janela do DOS. Dentro da janela do DOS, rode o programa *tabsis*. Você pode querer editar o programa e usar *fopen* para abrir um ou mais arquivos antes de exibir o conteúdo da tabela do sistema de arquivos.

DERIVANDO INDICATIVOS DE ARQUIVOS A PARTIR DE PONTEIROS STREAM (DE CANAIS) 374

A Dica 360 apresentou a estrutura *FILE* definida no arquivo de cabeçalho *stdio.h*. Você aprendeu, ao efetuar operações de arquivo de alto nível usando *fopen* e *fgets*, declarar ponteiros *stream* (de canais) em termos da estrutura *FILE*, como mostrado aqui:

```
FILE *entrada, *saida;
```

As funções C, mais tarde, convertem os ponteiros stream em indicativos de arquivos para efetuarem as operações reais de E/S. Para compreender melhor o relacionamento entre ponteiros stream e indicativos de arquivos, considere o programa a seguir, *indicat.c*, que abre arquivo *config.sys* do diretório-raiz e depois exibe o descritor de arquivo para o arquivo, bem como os indicativos de arquivo predefinidos *stdin*, *stdout*, *stderr*, *stdaux* e *stdprn*:

```
#include <stdio.h>

void main(void)
 {
   FILE *entrada;

   if ((entrada = fopen("\\CONFIG.SYS", "r")) == NULL)
      printf("Erro ao abrir \\CONFIG.SYS\N");
   else
    {
      printf("Indicativo para CONFIG.SYS %d\n", entrada->fd);
      printf("Indicativo para stdin %d\n", stdin->fd);
      printf("Indicativo para stdout %d\n", stdout->fd);
      printf("Indicativo para stderr %d\n", stderr->fd);
      printf("Indicativo para stdaux %d\n", stdaux->fd);
      printf("Indicativo para stdprn %d\n", stdprn->fd);
      fclose(entrada);
   }
 }
```

Quando você compilar e executar o programa *indicat.c*, sua tela exibirá o valores de indicativo de 0 até 5.

EXECUTANDO SAÍDA FORMATADA EM ARQUIVO 375

Várias dicas nesta seção apresentam modos de seus programas gravarem dados em um arquivo. Em muitos casos, seus programas precisam efetuar saída formatada em arquivo. Por exemplo, se você estiver criando um relatório de inventário, poderá querer alinhar as colunas, trabalhar com texto e com números e assim por diante. Na seção Introdução à Linguagem C, anteriormente, você aprendeu como usar a função *printf* para executar E/S formatada na tela. De um modo similar, C fornece a função *fprintf*, que usa especificadores de formato para gravar saída formatada em arquivo, como mostrado aqui:

```
#include <stdio.h>

int fprintf(FILE *pont_arquivo, const char *espec_formato, [argumento
[,...]]);
```

O programa a seguir, *fprintf.c*, usa *fprintf* para gravar saída formatada em um arquivo chamado *fprintf.dat*:

```
#include <stdio.h>

void main(void)
 {
   FILE *pa;

   int paginas = 800;
   float preco = 49.95;

   if (pa = fopen("FPRINTF.DAT", "w"))
     {
       fprintf(pa, "Título: Bíblia do Programador C/C++\n");
       fprintf(pa, "Páginas: %d\n", paginas);
       fprintf(pa, "Preço: $%5.2f\n", preco);
       fclose(pa);
     }
   else
     printf('Erro ao abrir FPRINTF.DAT\n");
 }
```

376 RENOMEANDO UM ARQUIVO

À medida que seus programas forem trabalhando com arquivos, algumas vezes você precisará renomear ou mover um arquivo. Para esses casos, C oferece a função *rename*. O formato da função *rename* é:

```
#include <stdio.h>

int rename(const char *nome_antigo, const char *nome_novo);
```

Se a função *rename* for bem-sucedida em renomear ou mover um arquivo, ela retornará o valor 0. Se um erro ocorrer, *rename* retornará um valor diferente de 0 e atribuirá à variável global *errno* um dos valores de status de erro listados na Tabela 376:

Tabela 376 Valores de status de erro para ***rename***.

Valor	Significado
EACCES	Acesso negado
ENOENT	Arquivo não-encontrado
EXDEV	Não pode mover de um disco para outro

O programa a seguir, *meu_ren.c*, usa a função *rename* para criar um programa que pode renomear ou mover o arquivo especificado na linha de comando:

```
#include <stdio.h>

void main(int argc, char *argv[])
 {
   if (argc < 3)
     printf("Você precisa especificar um nome de arquivo de origem e de
     destino\n");
   else if (rename(argv[1], argv[2]))
     printf('Erro ao renomear o arquivo\n");
 }
```

Nota: A Dica 1472 detalhará como renomear um arquivo usando a API do Windows.

EXCLUINDO UM ARQUIVO 377

Quando seus programas trabalham com arquivos, algumas vezes você precisa excluir um ou mais arquivos. Em tais casos, seus programas podem usar a função *remove* de C, que tem o seguinte formato:

```
#include <stdio.h>

int remove(const char *nomearq);
```

Se a função remover o arquivo com sucesso, ela retornará o valor 0. Se um erro ocorrer, *remove* retornará o valor -1, e atribuirá à variável global *errno* um dos valores listados na Tabela 377.

Tabela 377 Os erros que a função ***remove*** retorna

Valor	Significado
EACES	Acesso negado
ENOENT	Arquivo não-encontrado

O programa a seguir, *meu_del.c*, usa a função *remove* para apagar todos os arquivos especificados na linha de comando:

```
#include <stdio.h>

void main(int argc, char *argv[])
 {
   while (*++argv)
     if (remove(*argv))
       printf("Erro ao remover %s\n", *argv);
 }
```

Além da função *remove*, a maioria dos compiladores C suporta a função *unlink*, que também exclui um arquivo:

```
#include <io.h>

int unlink(const char *nomearq);
```

Se *unlink* excluir com sucesso um arquivo, ela retornará o valor 0. Se ocorrer um erro, *unlink* retornará o *status* de erro -1, atribuindo à variável global *errno* as constantes de status de erro já listadas na Tabela 377. O programa a seguir, *unlink.c*, usa a função *unlink* para excluir os arquivos especificados na linha de comandos do programa:

```
#include <stdio.h>

void main(int argc, char *argv[])
 {
   while (*++argv)
     if (unlink(*argv))
       printf("Erro ao remover %s\n", *argv);
 }
```

Nota: A Dica 1473 detalhará como excluir um arquivo usando a API do Windows.

DETERMINANDO COMO UM PROGRAMA PODE ACESSAR UM ARQUIVO 378

Quando seu programa trabalha com arquivos, algumas vezes você precisa determinar se seu programa pode acessar um arquivo específico. A função C *access* confere se um arquivo existe como especificado e se você pode abrir o arquivo conforme solicitado. O formato da função *access* é este:

```
#include <io.h>

int access(const char *nomearq, int modo_acesso);
```

O parâmetro *modo_acesso* especifica como seu programa precisa usar o arquivo, como mostrado na Tabela 378.1:

Tabela 378.1 Valores para o parâmetro ***modo_acesso***.

Valor	Significado
0	Verifica se o arquivo existe
2	Verifica se o arquivo pode ser gravado
4	Verifica se o arquivo pode ser lido
6	Verifica se o programa tem permissão de leitura e gravação no arquivo

Se o programa puder acessar o arquivo conforme especificado, *access* retornará o valor 0. Se um erro ocorrer, *access* retornará o valor -1, e atribuirá à variável global *errno* um dos valores de erro listados na Tabela 378.2.

Tabela 378.2 Valores de erro para a função ***access***.

Valor	Significado
EACCES	Acesso negado
ENOENT	Arquivo não-encontrado

O programa a seguir, *acesso.c*, usa a função *access* para determinar como seu programa pode acessar o arquivo especificado na linha de comando do programa:

```
#include <stdio.h>
#include <io.h>

void main(int argc, char *argv[])
 {
   int modo acesso;
   modo_acesso = access(argv[1], 0);

   if (modo_acesso)
     printf("O arquivo %s não existe\n");
   else
     {
       modo_acesso = access(argv[1], 2);
       if (modo acesso)
          printf("O arquivo não pode ser gravado\n");
       else
          printf("O arquivo pode ser gravado\n");
       modo_acesso = access(argv[1], 4);
       if (modo_acesso)
          printf("O arquivo não pode ser lido\n");
       else
          printf("O arquivo pode ser lido\n");
       modo_acesso = access(argv[1], 6);
       if (modo_acesso)
          printf("O arquivo não pode ser lido/gravado\n");
       else
          printf("O arquivo pode ser lido/gravado\n");
     }
 }
```

Nota: A Dica 1462 detalha como você usa os atributos de arquivos no Windows para determinar como um programa pode acessar um arquivo.

379 DEFININDO O MODO DE ACESSO DE UM ARQUIVO

Quando seus programas trabalham com arquivos, algumas vezes você desejará alterar o acesso de leitura e gravação de um programa. Por exemplo, assuma que você tenha um arquivo de dados importantes. Para proteger

o arquivo quando o programa não estiver em uso, você poderia definir o arquivo para o acesso de leitura somente. Desse modo, o usuário não pode excluir o arquivo acidentalmente. Quando o programa inicia, você pode modificar o arquivo para acesso de leitura e gravação, conforme for necessário. Para esses casos, seus programas podem usar a função *chmod* de C, como mostrado aqui:

```
#include <sys\stat.h>
#include <io.h>

int chmod (const char *nomearq, int modo_acesso);
```

O arquivo de cabeçalho *sys\stat.h* define as constantes de modo de acesso listadas na Tabela 379.1.

Tabela 379.1 Constantes de modo de acesso para ***chmod***.

Valor	Significado
S_IWRITE	A permissão de gravação está autorizada
S_IREAD	A permissão de leitura está autorizada

Para fornecer acesso de leitura e gravação, efetue uma operação *OU bit a bit* com as duas constantes (*S_IWRITE* | *S_IREAD*). Se *chmod* modificar com sucesso os atributos do arquivo, ela retornará o valor 0. Se ocorrer um erro, chmod retornará o valor -1 e definirá a variável global *errno* para um dos valores de status de erro listados na Tabela 379.2.

Tabela 379.2 Os valores de erro que *chmod* retorna.

Valor	Significado
ENOENT	Arquivo não-encontrado
EACCES	Permissão negada

O programa a seguir, *soleitur.c*, define o arquivo especificado na linha de comando para acesso de leitura somente:

```
#include <stdio.h>
#include <sys\stat.h>
#include <io.h>

void main(int argc, char *argv[])
 {
   if (chmod(argv[1], S_IREAD))
     printf("Erro ao definir %s\n", argv[1]);
 }
```

Nota: A Dica 1463 detalhará o uso de atributos de arquivos sob o Windows para mudar a maneira como um programa pode acessar um arquivo.

GANHANDO MELHOR CONTROLE DOS ATRIBUTOS DO ARQUIVO 380

Na dica anterior você aprendeu como usar a função *chmod* de C para definir os atributos de leitura e gravação de um arquivo. Quando você usar o sistema operacional DOS, poderá trabalhar com os atributos mostrados na Tabela 380.1:

Tabela 380.1 Os atributos que você pode usar com arquivos dentro do sistema operacional DOS.

Valor	Significado
FA_ARCH	Atributo de arquivar
FA_DIREC	Atributo de diretório
FA_HIDDEN	Atributo de oculto
FA_LABEL	Rótulo de volume do disco
FA_RDONLY	Atributo de leitura somente
FA_SYSTEM	Atributo de sistema

Nota: Alguns compiladores nomeiam essas constantes de forma diferente. Examine o arquivo dos.h fornecido com seu compilador, para conhecer os nomes corretos das constantes.

Para lhe ajudar a trabalhar com esses atributos, alguns compiladores C fornecem a função *_chmod*, cujo formato é mostrado aqui (lembre-se, os parâmetros mostrados dentro de colchetes são opcionais):

```
#include <dos.h>
#include <io.h>

int _chmod(const char *nomearq, int operacao [,int atributo]);
```

A operação diz a *_chmod* se você vai querer definir ou ler a definição do atributo. Se a função chamadora define a operação 1, *_chmod* define o atributo especificado. Portanto, o abre e fecha colchetes indicam que o parâmetro *atributos* é opcional. Se *_chmod* for bem-sucedida, ela retornará os atributos atuais do arquivo. Se ocorrer um erro, *_chmod* retornará o valor -1, e atribui à variável global *errno* um dos valores mostrados na Tabela 380.2.

Tabela 380.2 Os erros que ***_chmod*** retorna.

Valor	Significado
ENOENT	Arquivo não-encontrado
EACCES	Permissão negada

O programa a seguir, *atributo.c*, usa *_chmod* para exibir os atributos atuais do arquivo.

```
#include <stdio.h>
#include <dos.h>
#include <io.h>

void main(int argc, char *argv[])
 {
   int atributos;

   if ((atributos = _chmod(argv[1], 0)) == -1)
      printf("Erro ao acessar %s\n", argv[1]);
   else
     {
       if (atributos & FA_ARCH)
          printf('Arquivo ");
       if (atributos & FA_DIREC)
          printf('Diretório ");
       if (atributos & FA_HIDDEN)
          printf('Oculto ");
       if (atributos & FA_LABEL)
          printf('Rótulo do Volume ");
       if (atributos & FA_RDONLY)
          printf('Leitura somente ");
       if (atributos & FA_SYSTEM)
          printf('Sistema ");
     }
 }
```

Muitos compiladores C também fornecem a função *_dos_getfileattr* e *_dos_setfileattr*, que lhe permite ler ou definir os atributos do DOS, como mostrado aqui:

```
#include <dos.h>

int _dos_getfileattr(const char *nomearq, unsigned *atributos);
int _dos_setfileattr(const char *nomearq, unsigned *atributos);
```

As funções *_dos_getfileattr* e *_dos_setfileattr* usam as constantes de atributos detalhados na Tabela 380.3.

Tabela 380.3 As constantes de atributo que as funções *_dos_getfileattr* e *_dos_setfileattr* usam

Valor	Significado
_A_ARCH	Atributo Arquivar
_A_HIDDEN	Atributo Oculto
_A_NORMAL	Atributo Normal
_A_RDONLY	Atributo Leitura somente
_A_SUBDIR	Atributo Diretório
_A_SYSTEM	Atributo Sistema
_A_VOLID	*Rótulo de Volume do Disco*

Se as funções *_dos_getfileattr* e *_dos_setfileattr* são bem-sucedidas, elas retornam o valor 0. Se ocorre um erro, as funções retornam o valor -1, e atribuem à variável global *errno* o valor *ENOENT* (arquivo não-encontrado).

Como uma regra, seus programas somente devem manipular os atributos Arquivar, Leitura somente e Oculto, reservando os outros para uso pelo DOS. Se você somente modificar o atributo Leitura somente, use a função *chmod*, já apresentada na Dica 379, para aumentar a portabilidade do seu programa.

Nota: A Dica 1463 detalhará o uso de atributos de arquivo sob o Windows para modificar como um programa pode acessar um arquivo.

TESTANDO O ERRO NO CANAL DE UM ARQUIVO 381

Quando seus programas efetuam operações de E/S em arquivos, eles sempre devem testar os valores de retorno de funções, tais como, *fopen*, *fputs*, *fgets* e assim por diante para verificar se as operações foram bem-sucedidas. Para ajudar seus programas a realizar esse teste, C fornece a macro *ferror*, que examina o canal de E/S para verificar se houve um erro de leitura ou gravação. Se um erro ocorreu, *ferror* retorna um valor verdadeiro. Se nenhum erro ocorreu, *ferror* retorna falso, como mostrado aqui:

```
#include <stdio.h>

int ferror(FILE *canal);
```

Após ocorrer um erro de E/S em arquivo, a macro permanecerá verdadeira até que seus programas chamem a macro *clearerr* para o canal dado:

```
#include <stdio.h>

int clearerr(FILE *canal);
```

O programa a seguir, *ferror.c*, lê e exibe o conteúdo de um arquivo na tela. Após cada operação de E/S, o programa testa se houve um erro. Se ocorrer um erro, o programa terminará, exibindo uma mensagem de erro em *stderr*

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   FILE *pa;
```

```
  char linha[256];
  if (pa = fopen(argv[1], "r"))
   {
     while (fgets(linha, sizeof(linha), pa))
      {
        if (ferror(pa))
         {
           fprintf(stderr, "Erro ao ler de %s\n",argv[1]);
            exit(1);
         }
        else
         {
           fputs(linha, stdout);
           if (ferror(pa))
           {
             fprintf(stderr,
                        "Erro ao gravar em stdout\n");
             exit(1);
           }
         }
      }
   }
  else
    printf("Erro ao abrir %s\n", argv[1]);
}
```

382 . DETERMINANDO O TAMANHO DE UM ARQUIVO

À medida que seus programas realizarem operações de E/S em arquivo, algumas vezes você precisará determinar o tamanho em bytes de um arquivo. Para esses casos, você poderá usar a função *filelength* de C. A função *filelength* retorna um valor *long*. Você precisa passar ao programa um indicativo (*handle*) de arquivo, não um ponteiro de arquivo, como mostrado aqui:

```
#include <io.h>

long filelength(int indic_arquivo);
```

Se *filelength* for bem-sucedida, ela retornará o tamanho do arquivo em bytes. Se ocorrer um erro, *filelength* retornará o valor -1, e definirá a variável global *errno* como EBADF (número de arquivo inválido). O programa a seguir, *tamanho.c*, exibirá o tamanho do arquivo de um determinado arquivo na tela:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>

void main(int argc, char *argv[])
 {
   int indic_arquivo;
   long tamanho_arq;

   if ((indic_arquivo = open(argv[1], O_RDONLY)) == -1)
     printf("Erro ao abrir o arquivo %d\n", argv[1]);
   else
     {
       tamanho_arq = filelength(indic_arquivo);
       printf("O tamanho do arquivo em bytes é %ld\n", tamanho_arq);
       close(indic_arquivo);
     }
 }
```

Nota: A Dica 1463 detalhará como determinar o tamanho de um arquivo usando a API do Windows.

ESVAZIANDO UM CANAL DE E/S 383

Para aumentar o desempenho do seu programa, a biblioteca de execução de C normalmente bufferiza a saída do seu arquivo até que tenha um setor para gravar no disco, ou até fechar o arquivo. Deste modo, a biblioteca de execução reduz o número de operações lentas de E/S no disco. Infelizmente, quando seus programas usam um buffer assim, deixam a porta aberta para a possibilidade de perder dados. Quando seu programa efetua uma função, tal como *fputs*, para gravar saída e a função não retorna um erro, o programa assume que o sistema operacional gravou corretamente os dados no disco. No entanto, na realidade, os dados ainda podem residir na memória do seu computador Se o usuário desligar o computador, ele perderá dados. Se você tiver um programa para o qual precisa garantir que todos os dados sejam gravados no disco, poderá usar *fflush* para instruir a biblioteca de execução a gravar os dados de seu buffer na memória para o disco. O formato da função *fflush* é como segue:

```
#include <stdio.h>

int fflush(FILE *canal_arquivo);
```

Se *fflush* for bem-sucedida, ela retornará o valor 0. Se ocorrer um erro, *fflush* retornará a constante *EOF*. Os comandos a seguir ilustram como você pode usar *fflush* para esvaziar o buffer do arquivo no disco após cada operação de saída:

```
while (fgets(linha, sizeof(linha), arq_entrada))
 {
   fputs(linha, arq_saida);
   fflush(arq_saida);
 }
```

Nota: Quando você usa a função fflush, instrui a biblioteca de execução a chamar um serviço do sistema operacional para gravar os dados no disco. Se o sistema operacional efetuar sua própria bufferização (chamada cache do disco), ele poderá colocar seus dados em seu buffer de memória, e não no disco. Dependendo do software de cache no disco, talvez você possa chamar outro serviço do sistema para descarregar a saída.

FECHANDO TODOS OS ARQUIVOS ABERTOS DE UMA SÓ VEZ 384

Como discutido na Dica 361, antes de seus programas terminarem, você deve usar a função *fclose* para fechar seus arquivos abertos. Assuma que você tenha uma função que execute uma operação crítica. Se a função experimentar um erro, o programa deverá terminar de imediato. Infelizmente, a função pode não saber que existem arquivos abertos. Nesses casos, seu programa pode usar a função C *fcloseall* para fechar todos os arquivos abertos, como mostrado aqui:

```
#include <stdio.h>

int fcloseall(void);
```

Se *fcloseall* for bem-sucedida, ela retornará o número de arquivos que fechou. Se ocorreu um erro, *fcloseall* retorna a constante *EOF*. Os comandos a seguir ilustram como você poderia usar *fcloseall*:

```
if (status_erro == CRITICO)
 {
   fprintf(stderr, "Erro crítico de dispositivo\n");
   fcloseall();
   exit(1);
 }
```

OBTENDO O INDICATIVO DE ARQUIVO DE UM CANAL DE ARQUIVO 385

Como discutido na Dica 360, quando seus programas efetuam operações de arquivo, eles podem efetuar operações de baixo nível usando os canais de arquivo (*FILE *canal*). Você também pode usar os indicativos de ar-

quivo de baixo nível (*int indicativo*). Como foi visto, várias funções da biblioteca de execução de C requerem indicativos de arquivo. Se seu programa usa canais de arquivo, você pode fechar o arquivo e reabri-lo usando um indicativo de arquivo, ou pode obter um indicativo de arquivo usando a função *fileno* de C, como mostrado aqui:

```
#include <stdio.h>

int fileno(FILE *canal);
```

O programa a seguir, *fileno.c*, usa a função *fileno* para obter o indicativo de arquivo para um canal de arquivo aberto:

```
#include <stdio.h>
#include <io.h>

void main(int argc, char *argv[])
 {
   FILE *canal;
   int indicativo;
   long tamanho_arquivo;

   if (canal = fopen(argv[1], "r"))
     {
        // Alguns comandos
        indicativo = fileno(canal);
        tamanho_arquivo = filelength(indicativo);
        printf("O tamanho do arquivo é %ld\n", tamanho_arquivo);
        fclose(canal);
     }
   else
     printf("Erro ao abrir %s\n", argv[1]);
 }
```

386 CRIANDO UM NOME DE ARQUIVO TEMPORÁRIO USANDO P_TMPDIR

À medida que seus programas executam operações de E/S em arquivos, eles precisam abrir um ou mais arquivos temporários ou gravar saída em um arquivo não-existente no disco. Nesses casos, a dificuldade então torna-se determinar um nome de arquivo exclusivo para que o programa não sobrescreva um arquivo existente. Para auxiliar seus programas a gerar nomes de arquivos exclusivos, você pode usar a função *tmpnam*, como mostrado aqui:

```
#include <stdio.h>

char *tmpnam(char *buffer);
```

Se seu programa passar um buffer para *tmpnam*, a função atribuirá o nome temporário ao buffer. Se você chamar *tmpnam* com *NULL*, *tmpnam* alocará memória para o nome de arquivo, retornando ao programa um ponteiro para o início do nome de arquivo. A função *tmpnam* examina a entrada *P_tmpdir* no arquivo de cabeçalho *stdio.h*. Se *P_tmpdir* estiver definido, *tmpnam* cria o nome de arquivo exclusivo no diretório correspondente. Caso contrário, *tmpnam* criará o arquivo no diretório atual. Observe que *tmpnam* não cria na verdade o arquivo, mas em vez disso, retorna um nome de arquivo que seu programa pode usar com *fopen* ou *open*. O programa a seguir, *tmpnam.c*, ilustra o uso da função *tmpname*:

```
#include <stdio.h>

void main(void)
 {
   char buffer[64];
   int contador;

   for (contador = 0; contador < 5; contador++)
```

```
    printf("Nome de arquivo temporário %s\n", tmpnam(buffer));
}
```

Nota: O CD-ROM que acompanha este livro inclui o programa mak_temp.cpp, que cria um arquivo temporário com a API do Windows.

CRIANDO UM NOME DE ARQUIVO TEMPORÁRIO USANDO TMP OU TEMP 387

À medida que seus programas executam operações de E/S, eles freqüentemente precisam abrir um ou mais arquivos temporários ou gravar saída em um arquivo não-existente no disco. Nesses casos, a dificuldade então torna-se determinar um nome de arquivo exclusivo para que o programa não sobrescreva um arquivo existente. Para ajudar seus programas a gerar um nome de arquivo exclusivo, você pode usar a função *tempnam*, como mostrado aqui:

```
#include <stdio.h>

char *tempnam(char *buffer, char *prefixo);
```

Se seu programa passa um buffer para *tempnam*, a função atribui o nome temporário ao buffer. Se você chamar *tempnam* com *NULL*, *tempnam* alocará memória para o nome de arquivo, retornando para o programa um ponteiro para o início do nome de arquivo. O parâmetro *prefixo* lhe permite definir um conjunto de caracteres que você quer que *tempnam* coloque no início de cada nome de arquivo. A função *tempnam* examina as entradas do ambiente para determinar se uma entrada TMP ou TEMP existe. Se TMP ou TEMP for definido, *tempnam* criará o nome de arquivo exclusivo no diretório correspondente. Caso contrário, *tempnam* criará o arquivo no diretório atual. Observe que *tempnam* na verdade não cria o arquivo, mas, em vez disso, retorna um nome de arquivo que seu programa pode usar com *fopen* ou *open*. O programa a seguir, *tempnam.c*, ilustra o uso da função *tempnam*:

```
#include <stdio.h>

void main(void)
 {
   char buffer[64];
   int contador;

   printf("Nome de arquivo temporário %s\n", tempnam(buffer, "Biblia"));
 }
```

CRIANDO UM ARQUIVO VERDADEIRAMENTE TEMPORÁRIO 388

Nas Dicas 386 e 387 você aprendeu como usar as funções *tmpnam* e *tempnam* para gerar nomes de arquivo temporários. Como foi visto, *tempnam* e *tmpnam* na verdade não criam um arquivo, mas, em vez disso, elas simplesmente retornam um nome de arquivo que atualmente não está em uso. Além disso, C também fornece uma função chamada *tmpfile* que determina um nome de arquivo exclusivo, e, depois, abre o arquivo, retornando um ponteiro de arquivo para o programa. Você implementará a função *tmpfile* como mostrado aqui:

```
#include <stdio.h>

FILE *tmpfile(void);
```

Se *tmpfile* for bem-sucedida, ela abrirá o arquivo no modo de leitura e gravação, retornando um ponteiro de arquivo. Se ocorrer um erro, *tmpfile* retornará *NULL*. O arquivo que *tmpfile* retorna é um arquivo temporário. Quando o programa terminar (ou chamar *rmtemp*), o sistema operacional excluirá o arquivo e descartará seu conteúdo. Os comandos a seguir ilustram como seu programa poderia usar a função *tmpfile*:

```
FILE *arq_temp;

if (arq_temp = tmpfile())
 {
   // Arquivo temporário aberto com sucesso
```

```
    // Comandos que usam o arquivo
  }
else
    printf("Erro ao abrir o arquivo temporário\n");
```

389 REMOVENDO ARQUIVOS TEMPORÁRIOS

Na dica anterior você aprendeu que a função *tmpfile* permite que seus programas criem um arquivo temporário com conteúdo que existe somente durante a execução do programa. Dependendo dos seus programas, você poderá querer descartar os arquivos temporários antes de o *rd* programa terminar. Nesses casos, seu programa poderá usar a função *rmtmp*, cujo formato é mostrado aqui:

```
#include <stdio.h>
int rmtmp(void);
```

Se *rmtmp* for bem-sucedida, ela retornará o número de arquivo que fechou ou excluiu com sucesso.

390 PESQUISANDO O CAMINHO DE COMANDOS PARA UM ARQUIVO

Quando você trabalha dentro do ambiente do DOS, o comando PATH define os diretórios que o DOS pesquisa para os arquivos EXE, COM e BAT quando você executa o comando externo. Como os subdiretórios definidos no PATH em geral contêm seus comandos mais comumente usados, algumas vezes você poderá querer que um programa pesquise as entradas de subd.retório PATH para um arquivo de dados. Para esses casos, alguns compiladores fornecem a função *searchpath*. Você chama a função com o nome de arquivo desejado. Se *searchpath* localiza o arquivo com sucesso, ela retornará um nome de caminho completo para o arquivo que seus programas podem usar dentro de *fopen*. Se *searchpath* não encontrar o arquivo, ela retorna *NULL*, como mostrado aqui:

```
#include <dir.h>

char *searchpath(const char *nomearq);
```

O programa a seguir, *caminho.c*, ilustra o uso da função *searchpath* para pesquisar o nome de arquivo especificado:

```
#include <stdio.h>
#include <dir.h>

void main(int argc, char *argv[])
 {
   char *caminho;

   if (caminho = searchpath(argv[1]))
      printf("Nome do caminho: %s\n", caminho);
   else
      printf("Arquivo não-encontrado\n");
 }
```

Nota: A função searchpath pesquisa o arquivo especificado no diretório atual antes de pesquisar os subdiretórios indicados pelo comando Path.

391 PESQUISANDO UM ARQUIVO NA LISTA DE SUBDIRETÓRIO DE UM ITEM DO AMBIENTE

Na Dica 390 você usou a função *searchpath* para pesquisar um arquivo especificado nos diretórios no caminho de comandos. De um modo similar, você pode querer pesquisar os diretórios especificados em uma entrada de ambiente diferente para um arquivo. Por exemplo, muitos compiladores C definem as entradas *LIB* e *INCLUDE*, que especificam a localização dos arquivos de biblioteca (com a extensão *.lib*) e arquivos de cabeçalho (com a extensão *.h*).

Para pesquisar os diretórios que os itens *LIB* e *INCLUDE* especificam, você pode usar a função *_searchenv*, como mostrado aqui:

```
#include <dos.h>

char *_searchenv(const char *nomearq, const char *item_ambiente, *nomecaminho);
```

A função *_searchenv* pesquisa o nome de arquivo especificado nos diretórios especificados em *item_ambiente*. Se *_searchenv* encontrar o nome de arquivo, ela atribuirá o nome de caminho do arquivo ao buffer de string de caracteres do nome de caminho, retornando um ponteiro para o nome de caminho. Se *_searchenv* não encontrar o arquivo, ela retornará *NULL*. O programa a seguir, *pesq_amb.c*, usa a função *_searchenv* para pesquisar um arquivo especificado nos subdiretórios especificados no item *LIB*.

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   char caminho[128];

   _searchenv(argv[1], "LIB", caminho);
   if (caminho[0])
      printf("Nome do caminho: %s\n", caminho);
   else
      printf("Arquivo não-encontrado\n");
 }
```

Nota: A função _searchenv pesquisa no diretório atual o arquivo especificado antes de pesquisar os subdiretórios indicados no ambiente.

Nota: Nas Dicas 1474 até 1476 você aprenderá como usar a API do Windows para localizar arquivos no seu sistema.

ABRINDO ARQUIVOS NO DIRETÓRIO TEMP 392

Como você sabe, muitos programas criam seus arquivos temporários no subdiretório especificado pelo item TEMP do arquivo *config.sys*. Dentro de seus programas, você pode facilmente criar seus próprios arquivos dentro do diretório indicado pela linha TEMP, usando a função *getenv*. Os comandos a seguir ilustram como seus programas podem abrir um arquivo *tempdado.dat*, dentro do diretório temporário:

```
char nomecaminho[ MAX_CAMINHO];
strcpy(nomecaminho, getenv("TEMP"));

if (nomecaminho[0])
   strcat(nomecaminho, "\\TEMPDADO.DAT");
else
   strcat(nomecaminho, "TEMPDADO.DAT");
if (pa = fopen(nomecaminho, "w"))
```

Neste fragmento de código, se o item TEMP existir, o programa abrirá o arquivo no subdiretório correspondente. Se não houver um item TEMP, o programa abrirá o arquivo no diretório atual. Observe que o fragmento de código assume que a variável TEMP não contém um valor que termina com uma barra invertida. Idealmente, seus programas testarão o valor atual de TEMP e atuarão de forma apropriada.

MINIMIZANDO AS OPERAÇÕES DE E/S EM ARQUIVO 393

Comparado com a rápida velocidade eletrônica da CPU e da memória do seu computador, o disco mecânico é muito lento. Como resultado, você deve tentar minimizar o número de operações de E/S no disco que seus programas precisam executar. Com relação às operações de arquivo, o arquivo aberto provavelmente consome a maior parte do tempo. Portanto, você sempre deve examinar seus programas para garantir que não abra e feche um arquivo sem necessidade, ou que, repetidamente, abra um arquivo dentro de um laço. Por exemplo, considere os seguintes comandos:

```
while (escolha_menu != SAIR)
  {
    if (pa = fopen("BANCODAD.DAT", "r"))
      {
        // Pega o nome do cliente
        pega_cliente(nome);
        // Pesquisa no arquivo as informações do cliente
        pesquisa_info_cliente(nome, pa, buffer_dados);
        fclose(pa);
      }
    else
      {
         erro_abrir_arquivo("Abortando...");
      }
    escolha_menu = pega_escolha_menu();
  }
```

Os comandos repetem o laço, obtendo informações do cliente até que o usuário selecione a opção SAIR. Observe que a chamada da função *fopen* ocorre dentro do laço. Portanto, o programa repetidamente executa a operação lenta de E/S de arquivo. Para aumentar o desempenho do sistema, o programa deve colocar *fopen* fora do laço. Se a função *pesquisa_cliente* precisar começar no início do arquivo, o programa poderá rebobinar o arquivo, como mostrado aqui:

```
if (pa = fopen("BANCODAD.DAT", "r"))
    erro_abrir_arquivo("Abortando...");
while (escolha_menu != SAIR)
  {
     //.Pega o nome do cliente
     pega_cliente(nome);
     rewind(pa);
     // Pesquisa no arquivo as informações do cliente
     pesquisa_info_cliente(nome, pa, buffer_dados);
     escolha_menu = pega_escolha_menu();
  }
fclose (pa);
```

394 ESCREVENDO CÓDIGO QUE USA BARRAS INVERTIDAS NOS NOMES DE DIRETÓRIO

Várias dicas apresentadas nesta seção trabalham com nomes de diretório. Por exemplo, a função *chdir* permite que seus programas selecionem um diretório específico. Quando seu programa especifica um nome de diretório como um valor constante, certifique-se de usar barras invertidas duplas (\\) dentro de nomes de caminho, como requerido A seguinte chamada da função *chdir*, por exemplo, tenta selecionar o subdiretório DOS:

```
status = chdir("\DOS");
```

Quando você usa um caractere de barra invertida dentro de uma string, lembre-se de que C trata o caractere de barra invertida como um símbolo especial. Quando o compilador C encontra a barra invertida, ele confere o caractere que segue para determinar se é um símbolo especial, e, em caso afirmativo, substitui o caractere *com os correspondentes ASCII corretos. Se o caractere que aparece após a barra invertida não for um símbolo* especial, o compilador C ignorará o caractere da barra invertida. Portanto, a função *chdir* anterior tentaria selecionar o diretório DOS em vez de \\DOS. A chamada correta da função neste caso seria como segue:

```
status = chdir("\\DOS");
```

MODIFICANDO O DIRETÓRIO ATUAL 395

À medida que seus programas vão sendo executados, algumas vezes eles precisarão mudar do diretório atual. Para lhe ajudar a efetuar essa operação, a maioria dos compiladores fornece a função *chdir*. A função *chdir* é muito similar ao comando CHDIR do DOS: se você chamar a função com uma string que não contém uma letra de unidade de disco, *chdir* procurará o diretório na unidade atual. A seguinte chamada de função, por exemplo, seleciona o diretório *dados* na unidade C:

```
status = chdir("C:\\DADOS");   // Observe o uso de \\
```

De uma forma similar, o comando a seguir seleciona o diretório *tclite* na unidade atual:

```
status = chdir("\\TCLITE");
```

Se a função *chdir* for bem-sucedida, ela retornará o valor 0. Se o diretório não existir, *chdir* retornará o valor -1, e definirá a variável global *errno* com a constante *ENOENT*. O programa a seguir, *meuchdir.c*, implementa o comando CHDIR do DOS.

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <errno.h>

void main(int argc, char *argv[])
  {
    char diretorio[MAXPATH];

    if (argc == 1) // Exibe o diretório atual
      {
        getcwd(diretorio, MAXPATH);
        puts(diretorio);
      }
    else if ((chdir(argv[1])) && (errno == ENOENT))
      puts("Diretório inválido");
  }
```

Nota: Alguns compiladores definem o símbolo _MAX_PATH no arquivo de inclusão **direct.h** *(ou* **dir.h***), em vez de usar MAXPATH.*

Nota: A Dica 1468 explicará em detalhes a mudança de diretórios sob o Windows.

CRIANDO UM DIRETÓRIO 396

Durante a execução, seus programas podem precisar criar um diretório. Para ajudar seus programas a fazer isso, a maioria dos compiladores C fornece uma função *mkdir*, que é muito similar ao comando MKDIR do DOS. Se você chamar a função com uma string que não contém uma letra de unidade de disco, *mkdir* criará o diretório na unidade atual. Por exemplo, a seguinte chamada de função cria o diretório DADOS na unidade C.

```
status = mkdir("C:\\DADOS");    // Observe o uso de \\
```

De uma forma similar, o comando a seguir cria o diretório TEMPDADO na unidade atual, no diretório atual:

```
status = mkdir("TEMPDADO");
```

Se a função *mkdir* for bem-sucedida, ela retornará o valor 0. Se *mkdir* não puder criar o diretório, ela retornará o valor -1.

Nota: A Dica 1467 explicará em detalhes a criação de diretórios sob o Windows

397 REMOVENDO UM DIRETÓRIO

Durante a execução, seus programas podem precisar criar ou remover um diretório. Para ajudar seus programas a remover um diretório, a maioria dos compiladores C fornece a função *rmdir*, que é muito similar ao comando RMDIR do DOS. Se você chama a função com uma string que não contém uma letra de unidade de disco, *rmdir* cria o diretório na unidade atual. Por exemplo, a seguinte chamada de função remove o diretório DADOS na unidade C:

```
status = rmdir("C:\\DADOS");   // Observe o uso de \\
```

De uma forma similar, o comando a seguir remove o diretório TEMPDADO na unidade e no diretório atuais:

```
status = rmdir("TEMPDADO");
```

Se a função *rmdir* for bem-sucedida, ela retornará o valor 0. Se o diretório não existir ou *rmdir* não puder removê-lo, *rmdir* retornará o valor -1, e atribuirá à variável global *errno* um dos valores listados na Tabela 397.

Tabela 397 Os valores de erro para *mkdir*.

Valor	Significado
EACCES	Acesso negado
ENOENT	Esse diretório não existe

Nota: A Dica 1470 explicará em detalhes a remoção de diretórios sob o Windows.

398 REMOVENDO UMA ÁRVORE DE DIRETÓRIO

No MS-DOS versão 6, a Microsoft introduziu o comando DELTREE. DELTREE lhe permite, de uma só vez, excluir um diretório, seus arquivos e quaisquer subdiretórios dentro do diretório. Se você não usar o DOS versão 6, poderá criar seu próprio comando DELTREE usando o programa *deltree.c*, como mostrado aqui:

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
#include <alloc.h>
#include <string.h>

void main(int argc, char **argv)
 {
   void exclui_arvore(void);
   char buffer[128];
   char unidade[MAXDRIVE], diretorio[MAXDIR], nomearq[MAXFILE], ext[MAXEXT];

   if (argc < 2)
     {
       printf ("Erro de sintaxe\n");
       exit(0);
     }
   fnsplit (argv[1], unidade, diretorio, nomearq, ext);
   getcwd (buffer, sizeof(buffer));
   if (unidade[0] == NULL)
     {
       fnsplit (buffer, unidade, diretorio, nomearq, ext);
       strcpy (buffer, diretorio);
       strcat (buffer, nomearq);
       strcat (buffer, ext);
```

```
    }
  else
    {
     printf ("Não especifique a letra da unidade \n");
     exit (1);
    }
  if (strcmpi(buffer, argv[1]) == 0)
   {
    printf("Não é possível excluir o diretório atual\n");
    exit (1);
   }
  getcwd (diretorio, 64);
  if (chdir (argv[1]))
    printf ("Diretório inválido %s\n", argv[1]);
  else
    exclui_arvore();
  chdir (diretorio);
  rmdir (argv[1]);
}

union REGS regs_ent, regs_saida;
struct SREGS segs;

void exclui_arvore(void)
 {
  struct ffblk info_arq;
  int result;
  char far *farbuff;
  unsigned dados_seg, dados_desloc;

  result = findfirst("*.*", &info_arq, 16);
  regs_ent.h.ah = 0x2f;
  intdosx (&regs_ent, &regs_saida, &segs);
  dados_seg = segs.es;
  dados_desloc = regs_saida.x.bx;
  while (! result)
   {
    if ((info_arq.ff_attrib & 16) &&
          info_arq.ff_name[0] != '.')
     {
      regs_ent.h.ah = 0x1A;
      regs_ent.x.dx = FP_SEG(farbuff);
      segread(&segs);
      intdosx (&regs_ent, &regs_saida, &segs);
      chdir (info_arq.ff_name);
      exclui_arvore();
      chdir ("..");
      regs_ent.h.ah = 0x1A;
      regs_ent.x.dx = dados_desloc;
      segs.ds = dados_seg;
      rmdir (info_arq.ff_name);
    }
   else if (info_arq.ff_name[0] != '.')
    {
     remove (info_arq.ff_name);
```

```
    }
    result = findnext (&info_arq);
  }
}
```

Nota: O CD-ROM que acompanha este livro inclui o arquivo win_dire.cpp, que efetua a mesma tarefa que o programa deltree.c, mas que trabalha somente sob o Windows 95 ou sob o Windows NT.

399 CONSTRUINDO O NOME DE CAMINHO COMPLETO

Quando seus programas trabalham com arquivos e diretórios, você pode precisar conhecer o nome do caminho (completo) do arquivo. Por exemplo, se o diretório atual for *dados*, e a unidade atual for C, o nome completo do arquivo *relato.dat* será *c:\dados\relato.dat*. Para lhe ajudar a resolver o nome completo do arquivo (isto é, combinar seus componentes), alguns compiladores C fornecem uma função chamada *fnmerge*. A função usa cinco parâmetros: um buffer dentro do qual a função coloca o nome do caminho completo, o nome da unidade, o nome do diretório, o nome do arquivo e a extensão, como mostrado aqui:

```
#include <dir.h>

void fnmerge (char *buffer, const char *unidade, const char *dir, const
char *nomearq, const char *extensao);
```

Se o valor do parâmetro *buffer* for *NULL*, *fnmerge* alocará a memória usada para conter o nome de caminho completo. Se *fnmerge* resolver com sucesso o nome do arquivo, ela retornará um ponteiro para o buffer. Se ocorrer um erro, a função retornará *NULL*. O programa a seguir, *nomecomp.c*, ilustra o uso da função *fnmerge*:

```
#include <string.h>
#include <stdio.h>
#include <dir.h>

void main(void)
 {
    char s[MAXPATH];
    char unidade[MAXDRIVE];
    char dir[MAXDIR];
    char arquivo[MAXFILE];
    char ext[MAXEXT];

    getcwd(s,MAXPATH); // Obtém o diretório de trabalho atual
    strcat(s,"\\");  // Anexa um caractere \ preliminar
    fnsplit(s,unidade,dir,arquivo,ext);
  // Quebra a string em elementos
     separados
    strcpy(arquivo,"DADOS");
    strcpy(ext,".TXT");
    fnmerge(s,unidade,dir,arquivo,ext);
  // Mescla tudo em uma string
    puts(s); // Exibe string resultante
 }
```

Nota: Alguns compiladores usam o arquivo de inclusão direct.h, em vez de dir.h.

400 SUBDIVIDINDO UM CAMINHO DE DIRETÓRIO

À medida que seus programas trabalham com arquivos e diretórios, você pode precisar subdividir um nome de caminho em uma letra da unidade de disco caminho de subdiretório, nome de arquivo e extensão. Para lhe ajudar a subdividir um nome de caminho (isto é, separá-lo em seus componentes), alguns compiladores C fornecem a função *splitpath*. O formato da chamada da função é como segue:

```
#include <dir.h>
```

```
int fnsplit (const char *caminho, const char *unidade, const char
*diretorio, const char *nomearq, const char *ext);
```

O programa a seguir, *divide.c*, ilustra o uso da função *fnsplit*:

```
#include <stdio.h>
#include <direct.h>
#include <stdlib.h>

void main(void)
  {
    char *caminho_1 = "C:\\SUBDIR\\NOMEARQ.EXT";
    char *caminho_2 = "SUBDIR\\NOMEARQ.EXT";
    char *caminho_3 = "NOMEARQ.EXE";
    char subdir[MAXDIR];
    char unidade[MAXDRIVE];
    char nomearq[MAXFILE];
    char extensao[MAXEXT];
    int sinaliz;  // contém o valor de retorno de fnsplit

    sinaliz = fnsplit (caminho_1, unidade, subdir, nomearq, extensao);
    printf ('Dividindo %s\n", caminho_1);
    printf("Unidade %s Subdir %s Nomearq %s Extensão %s\n",
      unidade, subdir, nomearq, extensao);
    sinaliz = fnsplit (caminho_2, unidade, subdir, nomearq, extensao);
    printf ('Dividindo %s\n", caminho_2);
    printf("Unidade %s Subdir %s Nomearq %s Extensão %s\n",
      unidade, subdir, nomearq, extensao);
    sinaliz = fnsplit (caminho_3, unidade, subdir, nomearq, extensao);
    printf("Dividindo %s\n", caminho_3);
    printf("Unidade %s Subdir %s Nomearq %s Extensão %s\n",
      unidade, subdir, nomearq, extensao);
  }
```

Observe o uso das constantes para definir os tamanhos apropriados do buffer. Quando você compilar e executar o programa *split.c*, sua tela exibirá o seguinte:

```
Dividindo C:\SUBDIR\NOMEARQ.EXE
Unidade C: Subdir \SUBDIR\ Nomearq NOMEARQ Extensão .EXE
Dividindo \SUBDIR\NOMEARQ.EXE
Unidade  Subdir \SUBDIR\  Nomearq  NOMEARQ  Extensão .EXE
Dividindo  NOMEARQ.EXE
Unidade   Subdir   Nomearq  NOMEARQ   Extensão  .EXE
C:\>
```

CONSTRUINDO UM NOME DE CAMINHO 401

À medida que você trabalhar com arquivos e diretórios dentro de seus programas, algumas vezes poderá querer combinar uma letra de unidade de disco, subdiretório, nome de arquivo e extensão em um nome de caminho completo. Para lhe ajudar a efetuar essas operações, alguns compiladores C fornecem a função *fnmerge*. O formato da função *fnmerge* é como segue:

```
fnmerge(nomecaminho, unidade, subdir, nomearq, ext);
```

O programa a seguir, *criacam.c*, ilustra como usar a função *fnmerge*:

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>

void main(void)
 {
   char nomecaminho[MAXPATH];
```

```
    char *unidade = "C:";
    char *subdir = "\\SUBDIR";
    char *nomearq = "NOMEARQ";
    char *extensao = ".EXT";

    fnmerge(nomecaminho, unidade, subdir, nomearq, extensao);

    printf("O nome do caminho completo é %s\n", nomecaminho);
}
```

Quando você compilar e executar o programa *criacam.c*, sua tela exibirá a seguinte saída:

```
O nome do caminho completo é C:\SUBDIR\NOMEARQ.EXT
C:\>
```

402 ABRINDO E FECHANDO UM ARQUIVO USANDO FUNÇÕES DE BAIXO NÍVEL

C suporta operações de E/S de alto nível, que trabalham com canais (*streams*) de arquivo, e operações de baixo nível, que trabalham com intervalo de bytes. Quando seus programas efetuam E/S de baixo nível, você pode abrir um arquivo existente usando a função *open*. Para fechar o arquivo mais tarde, você usa *close*, como mostrado aqui:

```
#include <fcntl.h>
#include <sys\stat.h>

int open(const char *caminho, int modo_acesso [,modo_criac]);
int close(int close);
```

Se open abrir com sucesso o arquivo, retornará um indicativo para o arquivo. Se um erro ocorrer, open retornará -1, e definirá a variável global *errno* com um dos valores listados na Tabela 402.1.

Tabela 402.1 Códigos de status de erro que ***open*** atribui a ***errno***.

Valor	Significado
ENOENT	Não existe essa entrada de arquivo ou de diretório
EMFILE	Arquivos demais abertos
EACCES	Permissão de acesso negada
EINVACC	Código de acesso inválido

O parâmetro *caminho* é uma string de caracteres que contém o nome do arquivo desejado. O parâmetro *modo_acesso* especifica como você quer usar o arquivo. O valor *modo_acesso* pode ser uma combinação (use um operador *OU bit a bit*) dos valores listados na Tabela 402.2.

Tabela 402.2 Os valores possíveis para o parâmetro ***modo_acesso*** quando você o usa com ***open***.

Modo de Acesso	Significado
O_RDONLY	Acesso de leitura somente
O_WRONLY	Acesso de gravação somente
O_RDWR	Acesso de leitura e gravação
O_NDELAY	Valor de retardo que o UNIX usa
O_APPEND	Posiciona o ponteiro para as operações de anexação
O_TRUNC	Trunca o conteúdo de um arquivo existente
O_EXCL	Se *O_CREAT* está especificado e o arquivo já existe, *open* retorna um erro
O_BINARY	Abre o arquivo no modo binário
O_TEXT	Abre o arquivo no modo texto

Por padrão, *open* não criará um arquivo de saída se o arquivo não existir. Se você quiser que *open* crie arquivos, precisará incluir o sinalizador *O_CREAT* juntamente com os modos de acesso desejado (por exemplo, *O_CREAT* | *O_TEXT*). Se você especificar *O_CREAT*, poderá usar o parâmetro *modo_criacao* para especificar

o modo com o qual quer criar o arquivo. O parâmetro *modo_criacao* pode usar uma combinação dos valores que a Tabela 402.3 especifica.

Tabela 402.3 Os valores possíveis para o parâmetro ***modo_criacao*** usados por ***open***

Modo de Criação	Significado
S_IWRITE	Cria para operações de gravação
S_IREAD	Cria para operações de leitura

O comando a seguir ilustra como usar *open* para abrir o diretório-raiz *config.sys* para as operações de leitura somente:

```
if ((indicativo = open("\\CONFIG.SYS", O_RDONLY))== -1)
    printf("Erro ao abrir o arquivo \\CONFIG.SYS\n");
else
   // Comandos
```

Se você quiser abrir o arquivo *saida.dat* para operações de gravação e quiser que *open* crie um arquivo que ainda não existe, use *open* como segue:

```
if ((indicativo=open("\\CONFIG.SYS", O_RDONLY | O_CREAT, S_IWRITE)) == -1)
    printf("Erro ao abrir o arquivo \\CONFIG.SYS\n");
else
   // Comandos
```

Quando você terminar de usar um arquivo, deverá fechá-lo usando a função *close*, como mostrado aqui:

```
close(indicativo);
```

CRIANDO UM ARQUIVO 403

Na dica anterior você aprendeu que, por padrão, a função *open* não cria um arquivo se o arquivo não existe. No entanto, como foi visto, você poderá instruir *open* a criar um arquivo quando especifica *O_CREAT* no modo de acesso. Se você estiver usando um compilador mais antigo, a função *open* talvez não suporte *O_CREAT*. Como resultado, você pode precisar usar a função *creat*, como mostrado aqui:

```
#include <sys\stat.h>

int creat(const char *caminho, int modo_criacao);
```

Como antes, o parâmetro caminho especifica o arquivo que você quer criar. O parâmetro modo_criacao pode conter uma combinação dos valores listados na Tabela 403.

Tabela 403 Valores possíveis para o parâmetro ***modo_criacao***

Modo	Significado
S_IWRITE	Cria para operações de gravação
S_IREAD	Cria para operações de leitura

Se *creat* for bem-sucedida, ela retornará o indicativo para o arquivo. Se um erro ocorrer, *creat* retornará o valor -1, e atribuirá um valor de status de erro à variável global *errno*. O modo de tradução (binário ou texto) que *creat* usa depende da definição da variável global *_fmode*. Se um arquivo com o nome especificado existir, *creat* truncará o conteúdo do arquivo. O comando a seguir ilustra como usar *creat* para criar o arquivo *saida.dat*:

```
if ((indicativo = creat("SAIDA.DAT", S_IWRITE)) == -1)
   printf("Erro ao criar o arquivo\n");
else
   // Comandos
```

Nota: Se você quiser ser óbvio para outro programador que esteja criando um arquivo, poderá querer usar a função creat, em vez de usar open com o sinalizador O_CREAT ligado.

404 EFETUANDO OPERAÇÕES DE LEITURA E GRAVAÇÃO DE BAIXO NÍVEL

Ao usar indicativos de arquivos para efetuar operações de E/S de baixo nível em arquivos, você abre e fecha arquivos usando as funções *open* e *close*, como mostrado aqui:

```
#include <io.h>

int read(int indicativo, void *buffer, unsigned tamanho);
int write(int indicativo, void *buffer, unsigned tamanho);
```

O parâmetro *indicativo* é o indicativo que as funções *open* ou *creat* retornam. O parâmetro *buffer* é ou o buffer de dados no qual a função *read* lê informaçoes ou a partir de quais a função *write* grava dados. O parâmetro *tamanho* especifica o número de bytes que *read* ou *write* transferirá (o máximo é 65.535) Se *read* for bem-sucedida, ela retornará o número de bytes lidos. Se *read* encontrar o final do arquivo, *read* retornará 0. Por outro lado, *read* retorna -1, e define a variave. global *errno* para um dos valores listados na Tabela 404.

Tabela 404 Os valores de erro possíveis que ***read*** retorna.

Valor	Significado
EACCES	Acesso inválido
EBADF	Indicativo de arquivo inválido

Se *write* for bem-sucedida, ela retornará o número de bytes gravados. Se um erro ocorrer, *write* retornará o valor -1, e atribuirá à variável global *errno* um dos valores mostrados anteriormente. O laço a seguir ilustra como você poderia usar *read* e *write* para copiar o conteúdo de um arquivo para outro.

```
while ((bytes_lidos = read(entrada, buffer, sizeof(buffer))
   write(saida, buffer, bytes_lidos);
```

405 TESTANDO O FINAL DE UM ARQUIVO

Na dica anteriot vimos que a função *read* retorna o valor 0 quando encontra *EOF*. Dependendo do seu programa, algumas vezes você pode querer testar o final do arquivo antes de efetuar uma operação específica. Quando você usar indicativos de arquivo, a função *eof* retornará o valor 1 se o ponteiro de arquivo chegar ao final do arquivo, 0 se o ponteiro não estiver no final do arquivo, e -1 se o indicativo de arquivo for inválido·

```
#include <io.h>
int eof(int indicativo);
```

Os comandos a seguir modificam o código mostrado na Dica 404 para usar *eof* para testar o final do arquivo de entrada:

```
while (! eof(entrada))
 {
   bytes_lidos = read(entrada, buffer, sizeof(buffer));
   write(saida, buffer, bytes_lidos);
 }
```

406 COLOCANDO AS ROTINAS DE ARQUIVO DE BAIXO NÍVEL PARA TRABALHAR

Várias dicas nesta seção discutem as rotinas de E/S de arquivo de baixo nível. Para lhe ajudar a compreender melhor o uso de cada rotina, considere o programa a seguir, *copbaixa.c*, que usa as funções *read* e *write* para copiar o conteúdo do primeiro arquivo especificado na linha de comando para o segundo:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

void main(int argc, char *argv[])
 {
    int origem, destino;    // indicativos de arquivo
    char buffer[1024];      // buffer de E/S
    int bytes_lidos;

    if (argc < 3)
       fprintf(stderr, "É preciso especificar os arquivos de origem e de
destino\n");
    else if((origem = open(argv[1], O_BINARY | O_RDONLY)) == -1)
   fprintf(stderr, "Erro ao abrir %s\n", argv[1]);
    else if ((destino = open(argv[2], O_WRONLY | O_BINARY | O_TRUNC |
    O_CREAT, S_IWRITE)) == -1)
    fprintf(stderr, "Erro ao abrir %s\n", argv[2]);
    else
      {
         while (!eof(origem))
         {
           if ((bytes_lidos = read(origem, buffer, sizeof(buffer))) <= 0)
              fprintf(stderr, "Erro ao ler do arquivo de origem");
           else if(write(destino, buffer, bytes_lidos) != bytes_lidos)
           fprintf(stderr, "Erro ao gravar no arquivo de destino");
         }
         close(origem);
         close(destino);
      }
 }
```

ESPECIFICANDO O MODO PARA UMA TRADUÇÃO ARQUIVO-INDICATIVO 407

Como você aprendeu, C traduz o conteúdo de um arquivo usando a tradução binária ou texto. A não ser que você especifique de outra forma, C usa a definição na variável global *_fmode* para determinar o tipo de tradução, ou *O_BINARY* ou *O_TEXT*. Ao abrir ou criar um arquivo usando as rotinas de baixo nível de C, você poderá especificar o modo de tradução do arquivo. Em alguns casos, seu programa precisará especificar o modo de tradução após abrir o arquivo. Para especificar o modo, você poderá usar a função *setmode*, como mostrado aqui:

```
#include <fcntl.h>

int setmode(int indicativo, int modo_traducao);
```

Se *setmode* for bem-sucedida, ela retornará o modo de tradução anterior. Se ocorrer um erro, *setmode* retornará -1, e definirá a variável global *errno* como *EINVAL* (argumento inválido). Por exemplo, o comando a seguir, define o arquivo associado com o indicativo *saida* para tradução de texto:

```
if ((modo_antigo = setmode(saida, O_TEXT)) == -1)
    printf("Erro ao alterar o modo do arquivo\n");
```

408 POSICIONANDO O PONTEIRO DE ARQUIVO USANDO LSEEK

À medida que você trabalha com funções de E/S de arquivo de baixo nível, pode querer posicionar o ponteiro de arquivo em uma posição específica dentro do arquivo antes de efetuar uma operação de leitura ou gravação. Para fazer isso, você pode usar a função *lseek*, como mostrado aqui:

```
#include <io.h>

long lseek(int indicativo, long desloc, int relativo_a)
```

O parâmetro *handle* especifica o ponteiro de arquivo que você quer posicionar. Os parâmetros *desloc* e *relativo_a* combinam para especificar a posição desejada. O parâmetro *desloc* contém o byte de deslocamento dentro do arquivo. O parâmetro *relativo_a* especifica a posição no arquivo a partir do qual a função *lseek* deve aplicar *desloc*. A Tabela 408 especifica os valores que você pode usar para o parâmetro *relativo_a*.

Tabela 408 Posições no arquivo a partir das quais a função *lseek* pode aplicar um deslocamento

Constante	Significado
SEEK_CUR	A partir da posição atual no arquivo
SEEK_SET	A partir do início do arquivo
SEEK_END	A partir do final do arquivo

Por exemplo, para posicionar o ponteiro do arquivo no final de um arquivo, você pode usar *lseek* como segue:

```
lseek(indicativo, 0, SEEK_END);  // No final do arquivo
```

Se for bem-sucedida, *lseek* retornará o valor 0. Se ocorrer um erro, *lseek* retornará um valor diferente de 0.

409 ABRINDO MAIS DE 20 ARQUIVOS

Como você aprendeu, um indicativo de arquivo é um valor inteiro que identifica um arquivo aberto. Na verdade, um indicativo de arquivo é um índice em uma tabela de arquivo do processo, que contém entradas para até 20 arquivos. Se seu programa baseado no DOS precisar abrir mais do que 20 arquivos, o modo mais fácil para você poder fazer isso é usando os serviços de arquivo do DOS. Para começar, seu programa precisa solicitar suporte para mais de 20 arquivos. Você pode fazer isso usando a *função 67H* da interrupção *21H* do DOS para aumentar o número de indicativos de arquivos. O DOS então alocará uma tabela grande o bastante para conter o número de indicativos especificados (até 255 menos o número de indicativos atualmente em uso). Em seguida, seu programa deverá abrir os arquivos usando os serviços do DOS, em vez de usar a biblioteca de execução do DOS. Desse modo, seus programas poderão ignorar o limite de arquivo do compilador. O fragmento de código a seguir aumenta o número de indicativos de arquivo para 75:

```
inregs.h.ah = 0x67;
inregs.x.bx = 75;   // Número de indicativos
intdos(&regs_ent, &regs_saida);

if (regs_saida.x.ax)
    printf("Erro ao alocar indicativos\n");
```

Nota: O número de indicativos de arquivo disponíveis somente é importante no ambiente do DOS ou em uma janela do DOS. O Windows determina o limite do número de arquivos que você pode abrir ao mesmo tempo com base na memória atual, no espaço livre no disco e em outras considerações específicas do Windows.

410 USANDO SERVIÇOS DE ARQUIVO BASEADOS NO DOS

Como detalhado na seção Serviços DOS e BIOS, mais à frente, o DOS fornece uma coleção de serviços de arquivos que lhe permite abrir, gravar e fechar arquivos. Para tornar esses serviços mais fáceis de usar, muitos compiladores C fornecem as funções listadas na Tabela 410.

Tabela 410 Funções que usam o serviços do sistema de arquivos do DOS.

Função	Propósito
_dos_creat	Cria um arquivo, retornando um indicativo de arquivo
_dos_close	Fecha um arquivo especificado
_dos_open	Abre um arquivo, retornando um indicativo de arquivo
_dos_read	Lê o número especificado de bytes a partir de um arquivo
_dos_write	Grava o número especificado de bytes em um arquivo

Para lhe ajudar a compreender melhor os serviços de arquivo, considere o programa a seguir, *copiados.c*, que copia o conteúdo do primeiro arquivo especificado na linha de comando para o segundo arquivo especificado:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
 {
   char buffer[1024];
   int entrada, saida;    // indicativos de arquivos
   unsigned bytes_lidos, bytes_gravados; // número real de bytes transferidos

   if (argc < 3)
     fprintf(stderr, "Especifique os arquivos de origem e de destino\n");
   else if (_dos_open (argv[1], O_RDONLY, &entrada))
     fprintf(stderr, "Erro ao abrir o arquivo de origem\n");
   else if (_dos_creat (argv[2], 0, &saida))
     fprintf(stderr, "Erro ao abrir o arquivo de destino\n");
   else
    {
       while(!_dos_lidos(entrada, buffer, sizeof(buffer), &bytes_lidos))
       {
          if (bytes_lidos == 0)
             break;
          _dos write(saida, buffer, bytes_lidos, &bytes_gravados);
       }
       _dos_close(entrada);
       _dos_close(saida);
    }
 }
```

Nota: Embora as rotinas de arquivos baseadas no DOS sejam muito similares às funções de arquivo de baixo nível de C, você aumentará a portabilidade do seu programa se usar as funções **open**, **read** *e* **write** *de C, em vez de usar as funções baseadas no DOS. A maioria dos compiladores suporta as funções de baixo nível de C.*

Nota: Ao programar no Windows, você usará as funções da API do Windows em vez de usar rotinas de arquivo baseadas no DOS para gerenciar arquivos. As Dicas de 1450 até 1478 detalharão muitas funções de Arquivo da API do Windows.

OBTENDO A DATA E A HORA DE CRIAÇÃO DE UM ARQUIVO 411

Quando você efetuar uma listagem de diretório, o comando DIR do DOS exibirá o nome, extensão, tamanho, data e hora em que um arquivo foi criado ou alterado pela última vez. O DOS somente modifica a data e a hora quando você grava o arquivo. Por outro lado, alguns sistemas operacionais controlam a data e a hora em que o arquivo foi criado ou modificado pela última vez, bem como a data e a hora em que o arquivo foi usado (lido). Os sistemas operacionais referenciam essa segunda data e hora como *último horário de acesso*. Dependendo do propósito do seu programa, algumas vezes você precisará saber a data e a hora de criação/modificação de um arquivo. Portanto, a maioria dos compiladores fornece a função *_dos_getftime*, como mostrado aqui:

```
#include <dos.h>

unsigned _dos_getftime(int indicativo, unsigned *campodata, unsigned
*campohora)
```

Se a função ler com sucesso a data e a hora de um arquivo, ela retornará o valor 0. Se ocorrer um erro, a função retornará um valor diferente de 0, e atribuirá à variável global *errno* o valor *EBADF* (indicativo inválido). O parâmetro *indicativo* é um indicativo de arquivo aberto para o arquivo desejado. Os parâmetros *campodata* e *campohora* são ponteiros para valores inteiros sem sinal com significados de bits, como listados na Tabela 411.1 e 411.2, respectivamente.

Tabela 411.1 Os componentes do parâmetro ***campodada***

Bits de Data	Significado
0-4	Dia de 1 até 31
5-8	Mês de 1 até 12
9-15	Anos desde 1980

Tabela 411.2 Os componentes do parâmetro ***campohora***.

Bits de Data	Significado
0-4	Segundos divididos por 2 (de 1 até 30)
5-10	Minutos de 1 até 60
11-15	Horas de 1 até 12

O programa a seguir, *horario.c*, usa a função *_dos_getftime* para exibir a data e a hora do arquivo especificado na linha de comando:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
 {
   unsigned data, hora;
   int indicativo;

   if (_dos_open(argv[1], O_RDONLY, &indicativo))
      fprintf(stderr, "Erro ao abrir o arquivo de origem\n");
   else
    {
      if (_dos_getftime(indicativo, &data, &hora))
        printf("Erro ao ler a data/hora\n");
      else
          printf("%s última modificação %02d-%02d-%d %02d:%02d:%02d\n",
         argv[1],
         (data & 0x1E0) >> 5,   /* mês */
         (data & 0x1F),         /* dia */
         (data >> 9) + 1980,    /* ano */
         (hora >> 11),          /* horas */
         (hora & 0x7E0) >> 5,   /* minutos */
         (hora & 0x1F) * 2);    /* segundos */
      _dos_close(indicativo);
   }
 }
```

Como você pode ver, o programa usa os operadores bit a bit da linguagem C para extrair os campos *data* e *hora*. Na Dica 380 você aprendeu como efetuar processamento similar usando campos de uma estrutura de bits.

Nota: A dica 1465 detalhará como obter a data e a hora de um arquivo dentro do Windows.

OBTENDO A DATA E A HORA DE UM ARQUIVO USANDO CAMPOS DE BIT 412

Na dica anterior você usou a função *_dos_getftime* para obter a data e a hora de um arquivo. Como você aprendeu, a função *_dos_getftime* codifica os campos data e hora como bits dentro de dois valores sem sinal. Para extrair os valores do campo, o programa *horario.c* usa operadores bit a bit de C. Para tornar seu programa mais fácil de compreender, considere o uso de campos de bits dentro de uma estrutura. Para fazer isso, você pode usar o seguinte programa, *dhbits.c*:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
 {
   struct Date
    {
     unsigned int dia:5;
     unsigned int mes:4;
     unsigned int anos:7;
    } data;
   struct Time
    {
     unsigned segundos:5;
     unsigned minutos:6;
     unsigned horas:5;
    } hora;
   int indicativo;

   if (_dos_open(argv[1], O_RDONLY, &indicativo))
        fprintf(stderr, "Erro ao abrir o arquivo de origem\n");
    else
     {
        if (_dos_getftime(indicativo, &date, &time))
             printf("Erro ao ler a data e hora\n");
        else
            printf("%s última modificação %02d-%02d-%d %02d:%02d:%02d\n",
             argv[1],
             data.mes,              // mes
             data.dia,              // dia
             data.anos + 1980,      // ano
             hora.horas,            // horas
             hora.minutos,          // minutos
             hora.segundos * 2);    // segundos
        _dos_close(indicativo);
     }
 }
```

Usando campos de bits, o programa elimina a necessidade de outros programadores compreenderem as complicadas operações bit a bit que foram executadas no programa *horario.c*.

Nota: A Dica 1465 detalhará como obter a data e a hora de um arquivo sob o Windows.

DEFININDO A DATA E A HORA DE UM ARQUIVO 413

Nas Dicas 411 e 412 você usou a função *_dos_getftime* para ler a data e a hora de um arquivo. Dependendo do seu programa, você pode precisar definir a data e a hora de um arquivo. Para esses casos, muitos compiladores C fornecem a função *_dos_setftime*, como mostrado aqui:

```
#include <dos.h>
unsigned _dos_setftime(int indicativo, unsigned data, unsigned hora);
```

Se a função for bem-sucedida, ela retornará o valor 0. Se ocorrer um erro, a função retornará um valor diferente de 0. O parâmetro *indicativo* é um indicativo para um arquivo aberto. Os parâmetros data e hora contém os valores de data e hora codificados (similares aos mostrados na Dica 411). O programa a seguir, *maio2798.c*, define a data e a hora do arquivo especificado na linha de comando como meio-dia, 27 de maio de 1998:

```
#include <stdio.h>
#include <dos.h>
#include <fcntl.h>

void main(int argc, char *argv[])
 {
   union
    {
       struct Date
         {
           unsigned int dia:5;
           unsigned int mes:4;
           unsigned int anos:7;
         } bits;
       unsigned valor;
    } data;
   union
    {
      struct Time
        {
           unsigned segundos:5;
           unsigned minutos:6;
           unsigned horas:5;
        } bits;
       unsigned valor;
    } hora;
   int indicativo;

   if (_dos_open(argv[1], O_RDONLY, &indicativo))
        fprintf(stderr, "Erro ao abrir o arquivo de origem\n");
   else
      {
         data.bits.dia = 27;
         data.bits.mes = 5;
         data.bits.anos = 18;          // 1980 + 18
         hora.bits.horas = 12;
         hora.bits.minutos = 0;
         hora.bits.segundos = 0;
         if (_dos_setftime(indicativo, data.valor, hora.valor))
             printf("Erro ao definir a data/hora\n");
           _dos_close(indicativo);
      }
 }
```

O programa *maio2798.c* usa campos de bit para simplificar a atribuição dos bits da data e hora. No entanto, a função *_dos_setftime* requer parâmetros do tipo *unsigned int*. Como os bits precisam ser vistos em duas formas diferentes, eles são excelentes candidatos para uma união. A Dica 481 discute detalhadamente as uniões.

Nota: A Dica 1465 detalhará como definir a data e a hora de um arquivo sob o Windows.

DEFININDO A DATA E A HORA DE UM ARQUIVO COM A DATA E A HORA ATUAIS 414

Várias dicas aqui já mostraram modos de definir a data e a hora de um arquivo. Quando você quiser definir a data e a hora de um arquivo com a data e a hora atuais, poderá fazer isso rapidamente com a função *utime*, como mostrado aqui:

```
#include <utime.h>

int utime(char *caminho, struct utimbuf *data_hora);
```

O parâmetro *caminho* é uma string de caracteres que especifica o nome e o diretório do arquivo que você quer. O parâmetro *data_hora* é uma estrutura que contém a data e a hora em que o arquivo foi modificado e acessado pela última vez, como mostrado aqu.:

```
struct utimbuf
 {
   time_t actime;     // Último acesso
   time_t modtime;     // Última modificação
 };
```

Se você está trabalhando no ambiente do DOS, o DOS usa somente a hora de modificação. Se você chamar a função *utime* com *data_hora* definida como *NULL*, a função definirá a data e a hora com a data e a hora atuais. Se a função for bem-sucedida, ela retornará 0. Se ocorrer um erro, a função retornará -1, e definirá a variável global *errno*. O programa a seguir, *utime.c*, usa a função *utime* para definir a data e a hora do arquivo especificado com a data e a hora atuais:

```
#include <stdio.h>
#include <utime.h>

void main(int argc, char **argv)
 {
   if (utime(argv[1], (struct utimbuf *) NULL))
     printf("Erro ao definir a data e a hora\n");
   else
     printf("Data e a hora definidas\n");
 }
```

Nota: A Dica 1465 detalhará como definir a data e hora de um arquivo dentro do Windows.

LENDO E GRAVANDO DADOS UMA PALAVRA DE CADA VEZ 415

Como você aprendeu, as funções *getc* e *putc* lhe permitem ler e gravar informações do arquivo um byte de cada vez. Dependendo do conteúdo do seu arquivo, algumas vezes você poderá querer ler e gravar dados com uma palavra de cada vez. Para lhe ajudar a fazer isso, a maioria dos compiladores C fornece as funções *getw* e *putw*, como mostrado aqui:

```
#include <stdio.h>

int getw(FILE *canal);
int putw(int word, FILE *canal);
```

Se *getw* for bem-sucedida, ela retornará o valor inteiro lido do arquivo. Se um erro ocorrer ou *getw* encontrar o final do arquivo, ela retornará *EOF*. Se *putw* for bem-sucedida, retornará o valor inteiro que *putw* gravou no arquivo. Se ocorrer um erro, *putw* retornará *EOF*. O programa a seguir, *putwgetw.c*, usa a função *putwc* para gravar os valores de 1 a 100 em um arquivo. *Putwgetw.c* então abre o mesmo arquivo e lê os valores usando *getw*, como mostrado aqui:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
```

```
{
  FILE *pa;
  int palavra;

  if ((pa = fopen("DADOS.DAT", "wb")) == NULL)
    {
      printf("Erro ao abrir DADOS.DAT para saída\n");
      exit(1);
    }
  else
    {
      for (palavra = 1; palavra <= 100; palavra++)
         putw(palavra, pa);
      fclose(pa);
    }
  if ((pa = fopen("DADOS.DAT", "rb")) == NULL)
    {
       printf("Erro ao abrir DADOS.DAT para entrada\n");
       exit(1);
    }
  else
    {
      do
      {
        palavra = getw(pa);
        if ((palavra == EOF) && (feof(pa)))
           break;
        else
           printf("%d ", palavra);
      }
      while (1);
      fclose(pa);
    }
}
```

416 ALTERANDO O TAMANHO DE UM ARQUIVO

À medida que você trabalhar com arquivos, algumas vezes precisará alocar uma grande quantidade de espaço no disco para um arquivo ou desejará truncar o tamanho de um arquivo. Para esses casos, seus programas podem usar a função *chsize*, como mostrado aqui:

```
#include <io.h>

int chsize(int indicativo, long tamanho);
```

O parâmetro *indicativo* é o indicativo de arquivo que *open* ou *creat* retornaram anteriormente para o programa. O parâmetro *tamanho* especifica o tamanho de arquivo desejado. Se *chsize* for bem-sucedida, ela retornará o valor 0. Se ocorrer um erro, *chsize* retornará o valor -1 e definirá a variável global *errno* com um dos valores listados na Tabela 416.

Tabela 416 Valores de erro que ***chsize*** retorna.

Valor	Significado
EACCES	Acesso inválido
EBADF	Indicativo de arquivo inválido
ENOSPC	Espaço insuficiente (UNIX)

Se você aumentar o tamanho de um arquivo, então *chsize* preencherá o novo espaço do arquivo com caracteres *NULL*. O programa a seguir, *chsize.c*, criará um arquivo chamado *100zeros.dat*, e, depois, usa a função *chsize* para preencher com zeros os primeiros 100 bytes do arquivo:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>

void main(void)
 {
   int indic;

   if ((indic = creat("100ZEROS.DAT", S_IWRITE)) == -1)
         fprintf(stderr, "Erro ao abrir 100ZEROS.DAT");
   else
   {
     if (chsize(indic, 100L))
       printf("Erro ao modificar o tamanho do arquivo\n");
     close(indic);
   }
 }
```

CONTROLANDO AS OPERAÇÕES DE LEITURA E GRAVAÇÃO EM ARQUIVOS ABERTOS 417

Como foi visto, ao abrir um arquivo, independentemente se está usando *open*, *creat* ou *fopen*, você precisa especificar se quer acessar o arquivo no modo de leitura, de gravação ou leitura e gravação. A função *umask* lhe permite controlar como o programa mais tarde abre arquivos. O formato da função *umask* é como segue:

```
#include <io.h>

unsigned umask(unsigned modo_acesso);
```

O parâmetro *modo_acesso* especifica os modos que você quer impedir que o arquivo use. A Tabela 417 mostra os valores válidos para o parâmetro *modo_acesso*.

Tabela 417 Valores válidos para o parâmetro ***modo_acesso*** da função ***umask***.

Modo de Acesso	Significado
S_IWRITE	Impede o acesso de gravação
S_IREAD	Impede o acesso de leitura
S_IWRITE | *S_IREAD*	Impede o acesso de leitura e de gravação

Como um exemplo, se você quiser impedir que um programa abra arquivos com acesso para a gravação, usaria *umask*, assim:

```
modo_antigo = umask(S_IWRITE);
```

Como mostrado aqui, a função retorna a definição anterior. O programa a seguir, *umask.c*, usa a função *umask* para definir o modo de acesso como S_IWRITE, o que limpará o bit de acesso para a gravação do arquivo (tornando o arquivo de leitura somente). O programa então cria e grava saída no arquivo *saida.dat*. Após o programa fechar o arquivo, ele tentará abrir *saida.dat* para acesso de gravação. Como *umask* anteriormente definiu o arquivo como leitura somente, a operação de abertura falha, como mostrado aqui:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
```

```
#include <sys\stat.h>
#include <stdlib.h>

void main(void)
 {
   int saida;
   int def_antiga;

   def_antiga = umask(S_IWRITE);
   if ((saida = creat("SAIDA.DAT", S_IWRITE)) == -1)
     {
       fprintf(stderr, "Erro ao criar SAIDA.DAT\n");
       exit(1);
     }
   else
     {
       if (write(saida, "Teste", 4) == -1)
         fprintf(stderr, "Não é possível gravar no arquivo\n");
       else
         printf("Arquivo gravado com sucesso \n");
       close(saida);
     }
   if ((saida = open("SAIDA.DAT", O_WRONLY)) == -1)
     fprintf(stderr, "Erro ao abrir SAIDA.DAT para saida\n");
   else
     printf("Arquivo aberto para acesso de gravação\n");
 }
```

Nota: Para remover o arquivo saida.dat do seu disco, você precisa usar o comando ATTRIB -R saida.dat, e, depois, apagar o arquivo com o comando DEL do DOS.

418 ATRIBUINDO UM BUFFER DE ARQUIVO

Na seção Operações do Teclado, anteriormente, você aprendeu que C fornece funções de E/S que efetuam E/S bufferizada e direta. Para as operações de E/S bufferizadas, os dados são gravados ou lidos em um buffer antes de se tornarem disponíveis para seu programa. Por exemplo, as operações de arquivo usam E/S bufferizadas. Por outro lado, quando seus programas efetuam E/S direta, os dados ficam imediatamente disponíveis para seus programas sem serem colocados em um buffer intermediário. Normalmente você pode usar E/S direta para obter acesso ao teclado. Em geral, C aloca automaticamente um buffer para os canais de arquivo. No entanto, você pode usar a função *setbuf* para especificar seu próprio buffer, como mostrado aqui:

```
#include <stdio.h>

void setbuf(FILE *canal, char *buffer);
```

O parâmetro *canal* corresponde ao arquivo aberto ao qual você quer atribuir o novo buffer. O parâmetro *buffer* é um ponteiro para o buffer desejado. Se o parâmetro *buffer* contiver *NULL*, o arquivo aberto que o *canal* especifica não bufferizará os dados. O programa a seguir, *setbuf.c*, usa a função *setbuf* para alterar o buffer que C atribui para o indicativo de arquivo *stdout*. O programa então escreve saída em *stdout*. No entanto, como o programa está colocando os dados em um buffer maior, os dados não aparecem na tela antes de um retardo de três segundos. O programa então preenche o buffer com um caractere de cada vez, retardando dez milissegundos entre os caracteres. Quando o buffer ficar cheio, ele é descarregado (escrito) na tela, como mostrado aqui:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void main(void)
 {
   char buffer[512];
   int letra;
```

```
    setbuf(stdout, buffer);
    puts("Primeira linha de saída");
    puts("Segunda linha de saída");
    puts("Terceira linha de saída");
    delay(3000);
    printf("Prestes a preencher o buffer\n");
    fflush(stdout);
    for (letra = 0; letra < 513; letra++)
      {
       putchar('A');
       delay(10);
      }
}
```

ALOCANDO UM BUFFER DE ARQUIVO 419

Na dica anterior você aprendeu como usar a função *setbuf* para atribuir um buffer a um arquivo. Quando você usa *setbuf*, precisa especificar o buffer desejado. De um modo similar, muitos compiladores C fornecem a função *setvbuf*, que aloca um buffer (usando *malloc*) do tamanho desejado, e, depois, atribui o buffer ao arquivo especificado. Além disso, *setbuf* lhe permite especificar a bufferização que você deseja, como mostrado aqui:

```
#include <stdio.h>

int setvbuf(FILE *canal, char *buffer, int tipo_buffer, size_t tamanho);
```

O parâmetro *canal* é um ponteiro para um arquivo aberto. O parâmetro *buffer* é um ponteiro para o buffer no qual C bufferiza seus dados. Se o parâmetro *buffer* for *NULL*, a função *setvbuf* alocará o buffer para você. O parâmetro *tipo_buffer* lhe permite controlar o tipo do buffer. Finalmente, o parâmetro *tamanho* lhe permite especificar um tamanho de buffer até 32.767 bytes. Se *setvbuf* for bem-sucedida, ela retornará 0. Se ocorrer um erro (tal como memória insuficiente), *setvbuf* retornará um valor diferente de zero. A Tabela 419 lista os valores válidos para o parâmetro *tipo_buffer*.

Tabela 419 Tipos de bufferização válidos que *setvbuf* usa.

Tipo_Buffer	Bufferização
_IOFBF	Bufferização total. Quando o buffer estiver vazio, a próxima operação de leitura tentará preencher o buffer. Para saída, o buffer precisará estar cheio antes de *setvbuf* gravar dados no disco.
_IOLBF	Bufferização em linha. Quando o buffer estiver vazio, a próxima operação de leitura tentará preencher o buffer. Para saída, *setvbuf* gravará o buffer no disco quando o buffer estiver cheio ou quando encontrar o caractere de nova linha.
_IONBF	Sem bufferização. O programa efetuará E/S direta.

O programa a seguir, *setvbuf.c*, usa *setvbuf* para alocar um buffer de 8Kb para bufferização total

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void main(void)
 {
   char linha[512];
   char *buffer;
   FILE *entrada;

   if ((entrada = fopen("\\AUTOEXEC.BAT", "r")) == NULL)
      printf("Erro ao abrir \\AUTOEXEC.BAT\n");
   else
    {
      if (setvbuf(entrada, buffer, _IOFBF, 8192))
```

```
            printf("Erro ao modificar o buffer do arquivo\n");
         else
            while (fgets(linha, sizeof(linha), entrada))
              fputs(linha, stdout);
            fclose(entrada);
      }
   }
```

420 CRIANDO UM NOME DE ARQUIVO EXCLUSIVO USANDO MKTEMP

À medida que você trabalha com arquivos, a capacidade de criar um nome de arquivo exclusivo para os arquivos temporários é muito importante. Algumas das dicas nesta seção demonstram modos de criar nomes de arquivos aleatórios. Em muitos casos, você poderá querer criar um nome de arquivo exclusivo, mas também poderá querer que o nome de arquivo siga um formato específico que o relacione com o aplicativo. Por exemplo, para um programa de contabilidade, você poderá querer que todos os nomes de arquivos iniciem com as letras *CONTAB*. Para lhe ajudar a controlar a criação de nomes de arquivos exclusivos, muitos compiladores C fornecem a função *mktemp*, como mostrado aqui:

```
#include <dir.h>

char *mktemp(char *gabarito);
```

O *gabarito* é um ponteiro para uma string de caracteres que contém seis caracteres seguidos por seis Xs e um *NULL*. No caso do exemplo da contabilidade, *gabarito* seria um ponteiro para "CONTABXXXXXX". A função *mktemp* substitui os Xs com dois caracteres de nome de arquivo, um ponto e três caracteres para a extensão. Se *mktemp* for bem-sucedida, ela retornará um ponteiro para a string de gabarito. Se um erro ocorrer, a função retornará *NULL*. Como *mktemp* acrescenta letras ao parâmetro *gabarito*, você precisa alocar 13 ou mais posições de caracteres dentro da string. O programa a seguir, *mktemp.c*, ilustra como usar a função *mktemp*.

```
#include <stdio.h>
#include <dir.h>

void main(void)
 {
   char nome_a[13] = "CONTABXXXXXX";
   char nome_b[13] = "COMPUTXXXXXX";
   char nome_c[13] = "COMPRAXXXXXX";

   if (mktemp(nome_a))
     puts(nome_a);
   if (mktemp(nome_b))
     puts(nome_b);
   if (mktemp(nome_c))
     puts(nome_c);
 }
```

Quando você compilar e executar o programa *mktemp.c*, sua tela exibirá o seguinte:

```
CONTABAA.AAA
COMPUTAA.AAA
COMPRAAA.AAA
C:\>
```

421 LENDO E GRAVANDO ESTRUTURAS

A seção Matrizes, Ponteiros e Estruturas, mais à frente, apresenta muitos programas que trabalham com estruturas. Quando seus programas trabalham com estruturas, algumas vezes eles precisam armazenar os dados da estrutura em um disquete ou no disco rígido do computador e depois ler os dados. Como regra, quando você

precisar ler ou gravar uma estrutura, poderá tratá-la como um intervalo longo de bytes. Por exemplo, o programa a seguir, *dhsaida.c*, usa a função *write* de C para gravar a data e a hora atuais no arquivo *datahora.dat*.

```
#include <stdio.h>
#include <dos.h>
#include <io.h>
#include <sys\stat.h>

void main(void)
 {
   struct date data_atual;
   struct time hora_atual;
   int indicativo;

   getdate(&data_atual);
   gettime(&hora_atual);
   if ((indicativo = creat("DATAHORA.SAI",S_IWRITE)) == -1)
     fprintf(stderr, "Erro ao abrir o arquivo DATAHORA.SAI\n");
   else
     {
       write(indicativo, &data_atual, sizeof(data_atual));
       write(indicativo, &hora_atual, sizeof(hora_atual));
       close(indicativo);
     }
 }
```

Como você pode ver, para gravar a estrutura, o programa simplesmente passa o endereço da estrutura. De uma forma similar, o programa a seguir, *dhent.c*, usa a função *read* para ler as estruturas de data e hora:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>

void main(void)
 {
   struct date data_atual;
   struct time hora_atual;
   int indicativo;

   if ((indicativo = open("DATAHORA.SAI", O_RDONLY)) == -1)
     fprintf(stderr, "Erro ao abrir o arquivo DATAHORA.SAI\n");
   else
     {
       read(indicativo, &data_atual, sizeof(data_atual));
       read(indicativo, &hora_atual, sizeof(hora_atual));
       close(indicativo);
       printf("Data: %02d-%02d-%02d\n", data_atual.da_mon,
          data_atual.da_day, data_atual.da_year);
       printf("Hora: %02d:%02d\n", hora_atual.ti_hour,
          hora_atual.ti_min);
     }
 }
```

422 LENDO DADOS DE UMA ESTRUTURA A PARTIR DE UM CANAL DE ARQUIVO

Na dica anterior você aprendeu como usar as funções *read* e *urite* de C para efetuar operações de E/S em arquivos que usam estruturas. Se seus programas usam canais de arquivos, em vez de indicativos de arquivos, para E/S em arquivos você pode executar processamento similar usando as funções *fread* e *furite*, como mostrado aqui:

```
#include <stdio.h>

size_t fread(void *buffer, size_t tam_buffer, size_t conta_elemento, FILE
*stream)
size_t fwrite(void *buffer, size_t tam_buffer, size_t conta_elemento, FILE
*stream)
```

O parâmetro *buffer* contém um ponteiro para os dados que você quer gravar. O parâmetro *tam_buffer* especifica o tamanho em bytes dos dados. O parâmetro *conta_elemento* especifica o numero de estruturas que você está gravando, e o parâmetro *canal* é um ponteiro para um canal de arquivo aberto. Se as funções forem bem-sucedidas, retornarão o número de itens lidos ou gravados. Se ocorrer um erro ou uma das funções encontrar o final do arquivo, elas retornarão 0. O programa a seguir, *dhsa.c*, usa a função *furite* para gravar as estruturas de data e hora atuais em um arquivo:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct date data_atual;
   struct time hora_atual;
   FILE *saida;

   getdate(&data_atual);
   gettime(&hora_atual);
   if ((saida = fopen("DATAHORA.SAI", "w")) == NULL)
     fprintf(stderr, "Erro ao abrir o arquivo DATAHORA.SAI\n");
   else
     {
       fwrite(&data_atual, sizeof(data_atual), 1, saida);
       fwrite(&hora_atual, sizeof(hora_atual), 1, saida);
       fclose(saida);
     }
 }
```

Da mesma forma, o programa *dhentra.c* usa a função *fread* para ler os valores da estrutura, como mostrado aqui:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct date data_atual;
   struct time hora_atual;
   FILE *entrada;

   if ((entrada = fopen("DATAHORA.SAI", "r")) == NULL)
     fprintf(stderr, "Erro ao abrir o arquivo DATAHORA.SAI\n");
   else
     {
       fread(&data_atual, sizeof(data_atual),1, entrada);
       fread(&hora_atual, sizeof(hora_atual),1, entrada);
```

```
        fclose(entrada);
        printf("Data: %02d-%02d-%02d\n",data_atual.da_mon,
          data_atual.da_day, data_atual.da_year);
        printf("Hora: %02d:%02d\n", hora_atual.ti_hour, hora_atual.ti_min);
      }
}
```

DUPLICANDO O INDICATIVO DE UM ARQUIVO 423

Várias dicas nesta seção apresentaram funções que trabalham com indicativos de arquivos. Dependendo dos seus programas, algumas vezes você desejará duplicar o valor de um indicativo. Por exemplo, se seu programa efetua operações de E/S que são críticas, você pode querer duplicar o indicativo de arquivo, e, depois, fechar o novo indicativo copiado para descarregar a saída do arquivo para o disco. Como o primeiro indicativo de arquivo permanece aberto, você não tem a sobrecarga de reabrir o arquivo após a operação de descarga, como mostrado aqui:

```
#include <io.h>

int dup(int indicativo);
```

O parâmetro *indicativo* é o indicativo de arquivo aberto que você quer duplicar. Se *dup* duplicar com sucesso o indicativo, ela retornará um valor não-negativo. Se ocorrer um erro, *dup* retornará -1. O programa a seguir, *dup.c*, ilustra como você pode usar a função *dup* para descarregar os buffers de um arquivo.

```
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
#include <sys\stat.h>

void main(void)
 {
   int indic;
   int indic_duplicado;
   char titulo[] = "Bíblia do Programador C/C++";
   char secao[] = "Arquivos";

   if ((indic = open("OUTPUT.TST", O_WRONLY  O_CREAT, S_IWRITE)) == -1)
     printf("Erro ao abrir SAIDA.TST\n");
   else
    {
      if ((indic_duplicado = dup(indic)) == -1)
        printf("Erro ao duplicar o indicativo\n");
      else
       {
          write(indic, titulo, sizeof(titulo));
          close(indic_duplicado); // Descarrega o buffer
          write(indic, secao, sizeof(secao));
          close(indic);
       }
    }
}
```

FORÇANDO A DEFINIÇÃO DE UM INDICATIVO DE ARQUIVO 424

Na Dica 423 você aprendeu como usar o comando *dup* para criar uma cópia duplicada do conteúdo de um indicativo de arquivo. Algumas vezes você poderá querer alterar a definição de indicativo de um arquivo aberto e atribuir o valor de um indicativo diferente. Ao efetuar operações de modificação e atribuição com arquivos, você poderá usar *dup2*, como mostrado aqui:

```
#include <io.h>

int dup2(int indic_fonte, int indic_destino);
```

O parâmetro *indic_destino* é o indicativo de arquivo cujo valor você quer atualizar. Se a função atribuir com sucesso o indicativo, ela retornará o valor 0. Se ocorrer um erro, a função retornará -1. O parâmetro *indic_fonte* é o indicativo de arquivo cujo valor você quer atribuir ao destino. O programa a seguir, *dup2.c*, usa a função *dup2* para atribuir o valor da função *stderr* a *stdout*. Desse modo, os usuários não podem redirecionar a saída do programa da tela:

```
#include <stdio.h>
#include <io.h>

void main(void)
 {
   dup2(2, 1); // stdout é indicativo 1, stderr é indicativo 2
   printf("Esta mensagem não pode ser redirecionada!\n");
 }
```

425 ASSOCIANDO O INDICATIVO DE UM ARQUIVO COM UM CANAL

Muitas dicas nesta seção apresentam funções que trabalham com canais de arquivo ou com indicativos de arquivos. Dependendo do seu programa, algumas vezes você estará trabalhando com um indicativo de arquivo e irá querer usar uma função que corresponda a um canal de arquivo. Em tais casos, seus programas poderão usar a função *fdopen* para associar um indicativo de arquivo com um canal de arquivo, como mostrado aqui:

```
#include <stdio.h>

FILE *fdopen(int indicativo, char *modo_acesso);
```

O parâmetro *indicativo* é o indicativo de um arquivo aberto que você quer associar com um canal de arquivo. O parâmetro *modo_acesso* é um ponteiro para uma string de caracteres que especifica como você planeja usar o canal de arquivo. O valor *modo_acesso* precisa ser um dos valores de modo que você normalmente usaria com *fopen*. Se a função for bem-sucedida, ela retornará o ponteiro do canal. Se ocorrer um erro, a função retornará *NULL*. Por exemplo, o comando a seguir associa o indicativo de arquivo *entrada* com o ponteiro de arquivo *pa_ent* para acesso de leitura:

```
if ((pa = fdopen(entrada, "r")) == NULL)
   printf("Erro ao associar o arquivo\n");
else
  {
    gets(string, sizeof(string), pa_ent);
    fclose(pa_ent);
  }
```

426 COMPREENDENDO O COMPARTILHAMENTO DE ARQUIVO

Se você estiver trabalhando em um ambiente de rede e tiver o comando SHARE do DOS instalado, poderá escrever programas que permitam que mais de um programa acesse diferentes partes do mesmo arquivo ao mesmo tempo. Por exemplo, considere um programa que permita que múltiplos usuários reservem passagens em um avião. Quando um usuário quiser atribuir uma poltrona específica, o programa bloqueia aquela poltrona para que outro usuário não reserve a mesma poltrona. Após o programa atribuir a poltrona, o usuário a desbloqueia.

Ao compartilhar arquivos desse modo, você precisará usar primeiro a função *sopen* para abrir o arquivo para compartilhamento. Em seguida, quando seu programa quiser acessar um intervalo de bytes no arquivo, o programa tentará bloquear os dados. Se ninguém mais estiver usando (bloqueando) atualmente os dados, então o bloqueio do programa será bem-sucedido. Após o programa terminar com os dados, ele poderá desbloquear o intervalo de bytes no arquivo.

Quando um programa bloqueia um intervalo de bytes dentro de um arquivo, ele pode atribuir um bloqueio que permitirá que outros usuários acessem os dados de modos específicos. Por exemplo, o programa poderia deixar que outro arquivo lesse o intervalo bloqueado ou poderia permitir que outros programas lessem ou gravassem o mesmo intervalo de bytes. Várias das dicas a seguir discutem as funções de biblioteca de execução de C que suportam o compartilhamento e o bloqueio de arquivos.

ABRINDO UM ARQUIVO PARA O ACESSO COMPARTILHADO 427

Na dica anterior você aprendeu que pode usar o comando SHARE do DOS para abrir arquivos para múltiplos programas usarem ao mesmo tempo. Para abrir um arquivo para uso compartilhado, seus programas precisam usar a função *sopen*, como mostrado aqui:

```
#include <share.h>

int sopen(char *nomecaminho, int modo_acesso, int sinaliz_compart [, int
modo_criacao]);
```

Os parâmetros *nomecaminho*, *modo_acesso* e *modo_criacao* são similares aos usados pela função *open*. O parâmetro *sinaliz_compart* especifica como diferentes programas podem compartilhar o arquivo. Se *sopen* abrir com sucesso um arquivo, ela retornará um indicativo de arquivo. Se ocorrer um erro, *sopen* retornará -1. A Tabela 427 lista os valores válidos para o parâmetro *sinaliz_compart*.

Tabela 427 Os modos de acesso compartilhado que *sopen* suporta.

Sinalizador de Compartilhamento	Compartilhamento Permitido
SH_COMPAT	Permite o compartilhamento compatível
SH_DENYRW	Impede o acesso de leitura e gravação
SH_DENYWR	Impede o acesso de gravação
SH_DENYRD	Impede o acesso de leitura
SH_DENYNONE	Permite todos os acessos (leitura e gravação)
SH_DENYNO	Permite todos os acessos (leitura e gravação)

O programa a seguir, *sopen.c*, abre o arquivo especificado na linha de comando para acesso de leitura compartilhada. O arquivo então aguarda que você pressione uma tecla antes de ler e exibir o conteúdo do arquivo, como mostrado aqui:

```
#include <stdio.h>
#include <share.h>
#include <io.h>
#include <fcntl.h>

void main(int argc, char *argv[])
 {
   int indicat, bytes_lidos;
   char buffer[256];

   if ((indicat = sopen(argv[1], O_RDONLY,SH_DENYWR)) == -1)
      printf("Erro ao abrir o arquivo %s\n", argv[1]);
   else
     {
       printf("Pressione Enter para continuar \n");
       getchar();
       while (bytes_lidos = read(indicat, buffer, sizeof(buffer)))
         write(1, buffer, bytes_lidos);  // 1 é stdout
       close(indicat);
     }
 }
```

Para compreender melhor como o programa *sopen.c* funciona, chame o comando SHARE. Em seguida, inicie o Windows e crie uma janela do DOS dentro da qual você roda o programa usando o nome de arquivo *sopen.c* como o arquivo compartilhado. Quando o programa pedir que você pressione uma tecla, abra uma segunda janela do DOS e use TYPE para exibir o conteúdo do arquivo. Enquanto TYPE exibe o conteúdo do arquivo *sopen.c*, dois programas têm o arquivo aberto ao mesmo tempo. Feche a janela e retorne à primeira janela. Pressione Enter para exibir o conteúdo do arquivo. Experimente com o programa *sopen.c* encadeando os modos de compartilhamento. Repita o processo de tentar acessar o arquivo usando dois programas.

428 BLOQUEANDO O CONTEÚDO DE UM ARQUIVO

Como você aprendeu, ao compartilhar o conteúdo de um arquivo, algumas vezes você desejará bloquear um intervalo de bytes dentro de um arquivo para impedir que outro programa os modifique. Para bloquear um intervalo específico de bytes dentro de um arquivo, seus programas podem usar a função *lock*, como mostrado aqui:

```
#include <io.h>

int lock(int indicativo, long pos_inicial, long conta_byte);
```

O parâmetro indicativo é um indicativo que corresponde a um arquivo que *sopen* abriu para compartilhamento. O parâmetro *pos_inicial* especifica o número de bytes que você quer bloquear. Se a função *lock* bloquear com sucesso o intervalo de bytes, ela retornará o valor 0. Se ocorrer um erro, a função retornará -1. Você precisará ter o comando SHARE do DOS instalado para a função *lock* funcionar.

Após você bloquear um intervalo de bytes, outros programas tentarão três vezes ler ou gravar o intervalo bloqueado. Se, após a terceira tentativa, o programa não puder ler os dados, então a função *read* ou *write* retornará um erro. O programa a seguir, *blocauto.c*, bloqueia os primeiros cinco bytes do arquivo *autoexec.bat*, e, depois, aguarda que você pressione uma tecla:

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>

void main(void)
 {
   int indicat;

   if ((indicat = sopen("\\AUTOEXEC.BAT", O_RDONLY, SH_DENYNO)) == -1)
     printf("Erro ao abrir AUTOEXEC.BAT\n");
   else
    {
      lock(indicat, 0L, 5L);
      printf("Arquivo bloqueado -- pressione  Enter para continuar \n");
      getchar();
      close(indicat);
    }
 }
```

Depois, o seguinte programa, *tentauto.c*, tenta ler o arquivo *autoexec.bat*, um byte de cada vez. Se ocorrer um erro enquanto ele lê o arquivo, então o programa exibirá uma mensagem de erro, como mostrado aqui:

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>

void main(void)
 {
   int indicat;
   int desloc = 0;
   int bytes_lidos;
   char buffer[128];

   if ((indicat = sopen("\\AUTOEXEC.BAT", O_BINARY | O_RDONLY, SH_DENYNO))
   == -1)
     printf("Erro ao abrir AUTOEXEC.BAT\n");
   else
    {
      while (bytes_lidos = read(indicat, buffer, 1))
       {
         if (bytes_lidos == -1)
```

```
            printf("Erro ao ler o deslocamento %d\n", desloc);
          else
            write(1, buffer, bytes lidos);
          desloc++;
          lseek(indicat, desloc, SEEK_SET);
        }
      close(indicat);
    }
}
```

GANHANDO CONTROLE MAIS REFINADO DO BLOQUEIO DE ARQUIVO 429

Na dica anterior você aprendeu como usar a função *lock* para bloquear um intervalo de bytes dentro de um arquivo. Ao usar a função *lock*, a operação será bem-sucedida ou falhará imediatamente. Se você quiser controle *mais refinado da operação de bloqueio, poderá usar a função locking, como mostrado aqui:*

```
#include <io.h>
#include <sys\locking.h>

int locking(int indicativo, int comando_bloqueio, long conta_byte);
```

O parâmetro *indicativo* é o indicativo associado com o arquivo que você quer bloquear. O parâmetro *comando_bloqueio* especifica a operação de bloqueio desejada. O parâmetro *conta_byte* especifica o número de bytes que você quer bloquear. O início da região depende do ponteiro da posição atual do arquivo. Se você quiser bloquear uma região específica, poderá primeiro usar a função *lseek* para posicionar o ponteiro de arquivo. A Tabela 429.1 especifica os valores possíveis para *comando_bloqueio*.

Tabela 429.1 Comandos que a função ***locking*** utiliza.

Comando de Bloqueio	Significado
LK_LOCK	Bloqueia a região especificada. Se o bloqueio não for bem-sucedido, *locking* tentará uma vez todo segundo, por dez segundos, aplicar o bloqueio.
LK_RLCK	Efetua as mesmas funções que *LK_LOCK*
LK_NBLCK	Bloqueia a região especificada. Se o bloqueio não for bem-sucedido, *locking* retornará um erro imediatamente.
LK_UNLCK	Desbloqueia uma região anteriormente bloqueada.

Se a função *locking* bloquear com sucesso o arquivo, ela retornará o valor 0. Se ocorrer um erro, a função *locking* retornará o valor -1, e definirá a variável global *errno* com um dos valores especificados na Tabela 429.2.

Tabela 429.2 Valores de status de erro que a função ***locking*** retorna.

Erro de Status	Significado
EBADF	Indicativo de arquivo inválido
EACCESS	Arquivo já bloqueado ou desbloqueado
EDEADLOCK	O arquivo não pode ser bloqueado após 10 tentativas
EINVAL	Comando inválido especificado

O programa a seguir, *locking.c*, altera o programa *blocauto.c* já apresentado na Dica 428 para usar a função *locking* para bloquear os primeiros cinco bytes do arquivo *autoexec.bat:*

```
#include <stdio.h>
#include <io.h>
#include <share.h>
#include <fcntl.h>
#include <sys\locking.h>
void main(void)
```

```
{
   int indicat;

   if ((indicat = sopen("\\AUTOEXEC.BAT", O_RDONLY, SH_DENYNO)) == -1)
      printf("Erro ao abrir AUTOEXEC.BAT\n");
   else
    {
      printf("Tentando bloquear o arquivo \n");
      if (locking(indicat, LK_LOCK, 5L))
         printf("Erro ao bloquear o arquivo\n");
      else
       {
         printf("Arquivo bloqueado--pressione Enter para continuar\n");
         getchar();
         close(indicat);
       }
    }
}
```

Como antes, se você tiver o Windows disponível, experimente rodar o programa *locking.c* dentro de duas janelas do DOS ao mesmo tempo.

Nota: Antes de você usar a função locking, instale o comando SHARE do DOS.

430 Trabalhando com Diretórios do DOS

Dentro de seus programas, você pode usar as funções *findfirst* e *findnext* para trabalhar com arquivos que atendam a uma combinação específica de caractere-chave (por exemplo, *"*.exe"*). Como o DOS não trata os diretórios como arquivos, seus programas não podem usar os serviços do DOS para "abrir" um diretório e ler seu conteúdo. No entanto, se você compreende como o DOS dispõe as informações no disco, seus programas podem ler a partir da tabela de alocação de arquivos e do diretório-raiz, e, depois, ler e controlar os setores que contêm as entradas de um diretório. Os comandos utilitários de disco (tais como UNDELETE) e as ferramentas de ordenação de diretórios efetuam essas operações de E/S de disco de baixo nível. Várias dicas a seguir ilustram como seus programas podem usar essas funções de E/S de diretório. Para simplificar a tarefa de ler um diretório, alguns compiladores C fornecem as funções listadas na Tabela 430.

Tabela 430 Funções de E/S de diretório e seus propósitos.

Função	Propósito
closedir	Fecha um canal de diretório
opendir	Abre um canal de diretório para operações de leitura
readdir	Lê a próxima entrada em um canal de diretório
rewinddir	Move o ponteiro do canal de diretório de volta para o início do diretório

431 Abrindo um Diretório

Na Dica 430 você aprendeu que muitos compiladores C fornecem funções que lhe permitem abrir e ler os nomes de arquivos que residem em um diretório específico. Para abrir um diretório para as operações de leitura, seus programas podem usar a função *opendir*, como mostrado aqui:

```
#include <dirent.h>

DIR *opendir(char *nome_diretorio);
```

O parâmetro *nome_diretorio* é um ponteiro para uma string de caracteres que contém o nome de diretório desejado. Se o nome de diretório for *NULL*, *opendir* abrirá o diretório atual. Se a função *opendir* for bem-sucedida, ela retornará um ponteiro para uma estrutura do tipo *DIR*. Se um erro ocorrer, a função retornará *NULL*. Por exemplo, o comando a seguir ilustra como você poderia abrir o diretório DOS para operações de leitura:

```
struct DIR *entrada_diretorio;

if ((entrada_diretorio = opendir("\\DOS")) == NULL)
    printf("Erro ao abrir o diretorio\n");
else
   // Comandos
```

Após você ter efetuado suas operações de leitura de diretório, deverá fechar o canal de diretório usando a função *closedir*, como mostrado aqui:

```
#include <dirent.h>

void closedir(DIR *diretorio);
```

LENDO UMA ENTRADA DO DIRETÓRIO 432

Na Dica 431 você viu como usar a função *opendir* para abrir uma listagem de diretório. Após você abrir um diretório, poderá usar a função *readdir* para ler o nome da próxima entrada na lista do diretório, como mostrado aqui:

```
#include <dirent.h>

struct dirent readdir(DIR *ponteiro_diretorio);
```

O parâmetro *ponteiro_diretorio* é o ponteiro que a função *opendir* retorna. Se *readdir* ler com sucesso uma entrada de diretório, ela retornará a entrada lida. Se ocorrer um erro ou *readdir* chegar ao final do diretório, a função retornará *NULL*. A função *readdir* lê todas as entradas na lista do diretório, incluindo as entradas "." e "..".

USANDO SERVIÇOS DE DIRETÓRIO PARA LER C:\WINDOWS 433

Na Dica 431 você aprendeu como abrir e fechar uma listagem de diretório. Na Dica 432, como usar a função *readdir* para ler a próxima entrada na lista de diretório. O programa a seguir, *exibedir.c*, usa a biblioteca de execução para ler, abrir e depois fechar o diretório especificado na linha de comando:

```
#include <stdio.h>
#include <dirent.h>

void main(int argc, char *argv[])
 {
   DIR *ponteiro_diretorio;
   struct dirent *item;

   if ((ponteiro_diretorio = opendir(argv[1])) == NULL)
      printf("Erro ao abrir %s\n", argv[1]);
   else
     {
        while (item = readdir(ponteiro_diretorio))
           printf("%s\n", item);
        closedir(ponteiro_diretorio);
     }
 }
```

Por exemplo, o comando a seguir usa o programa *exibedir.c* para exibir os nomes dos arquivos no diretório *c:\windows*:

```
C:\> EXIBEDIR  C:\WINDOWS   <Enter>
```

VOLTANDO PARA O INÍCIO DE UM DIRETÓRIO 434

Na Dica 433 você aprendeu que C fornece funções da biblioteca de execução que lhe permitem abrir e ler os nomes de arquivos em um diretório especificado. À medida que você for lendo diretórios, algumas vezes poderá querer começar a leitura dos arquivos no início da lista de diretórios pela segunda vez. Um modo de executar essa operação é fechar e depois reabrir a lista de diretório. Alternativamente, seus programas podem usar a função *rewinddir*, como mostrado aqui:

```
#include <dirent.h>

void rewinddir(DIR *ponteiro_diretorio);
```

O parâmetro ponteiro_diretorio é o ponteiro para a lista de diretório que você quer reinicializar. Se você experimentar essa função rewinddir, verá que ela é muito mais fácil que fechar e reabrir a lista de diretório.

435 Lendo Recursivamente os Arquivos de um Disco

Na Dica 433 você usou o programa *exibedir.c* para exibir os arquivos em uma lista de diretórios. O programa a seguir, *todosarq.c*, usa as funções da biblioteca de execução para exibir os nomes de todos os arquivos no seu disco. Para fazer isso, o programa usa a função recursiva *exibe_dir* para exibir os nomes de arquivos, como mostrado aqui:

```
#include <stdio.h>
#include <dirent.h>
#include <dos.h>
#include <io.h>
#include <direct.h>
#include <string.h>

void exibe_dir(char *nome_dir)
 {
   DIR *ponteiro_diretorio;
   struct dirent *item;
   unsigned atributos;

   if ((ponteiro_diretorio = opendir(nome_dir))== NULL)
      printf("Erro ao abrir %s\n", nome_dir);
   else
   {
        chdir(nome_dir);
        while (item = readdir(ponteiro_diretorio))
          {
            atributos = _chmod(item, 0);
            // Verifica se o item é para um subdiretório e não é "." ou ".."
            if ((atributos & FA_DIREC) &&
               (strncmp(item, ".", 1) != 0))
             {
               printf("\n\n----%s----\n", item);
               exibe_dir(item);
             }
            else
             printf("%s\n", item);
          }
        closedir(ponteiro_diretorio);
        chdir("..");
   }
 }

void main(void)
 {
    char buffer[MAXPATH];

    // Salva o diretório atual para que você possa restaurá-lo depois
    getcwd(buffer, sizeof(buffer));
    exibe_dir("\\");
    chdir(buffer);
 }
```

DETERMINANDO A POSIÇÃO ATUAL NO ARQUIVO 436

Você aprendeu anteriormente como C controla a posição atual em arquivos que estão abertos para operações de entrada e saída. Dependendo do seu programa, algumas vezes você precisa determinar o valor do ponteiro de posição. Se você estiver trabalhando com canais de arquivo, poderá usar a função *ftell* para determinar a posição do ponteiro de arquivo. No entanto, se você estiver trabalhando com indicativos de arquivo, seus programas poderão usar a função *tell*, como mostrado aqui:

```
#include <stdio.h>

long tell(int indicativo);
```

A função *tell* retorna um valor *long* que especifica o deslocamento de byte da posição atual no arquivo especificado. O programa a seguir, *tell.c*, usa a função *tell* para exibir informações do ponteiro de posição. O programa inicia abrindo o arquivo *config.sys* do diretório-raiz no modo de leitura. Em seguida, o programa usa *tell* para exibir a posição atual. Em seguida, o programa lê e exibe o conteúdo do arquivo. Após o programa encontrar o final do arquivo, ele novamente usa *tell* para exibir a posição atual, como mostrado aqui:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>

void main(void)
 {
   int indic;
   char buffer[512];
   int bytes_lidos;

   if ((indic = open("\\CONFIG.SYS", O_RDONLY)) == -1)
     printf("Erro ao abrir \\CONFIG.SYS\n");
   else
     {
       printf("Posição atual no arquivo %ld\n", tell(indic));
       while (bytes_lidos = read(indic, buffer, sizeof(buffer)))
         write(1, buffer, bytes_lidos);
       printf("Posição atual no arquivo %ld\n", tell(indic));
       close(indic);
     }
 }
```

ABRINDO UM CANAL DE ARQUIVO COMPARTILHADO 437

Várias dicas nesta seção apresentam modos de compartilhar e bloquear os arquivos usando indicativos de arquivo. Se você normalmente trabalha com canais de arquivo, seus programas podem usar a função *_fsopen*, como mostrado aqui:

```
#include <stdio.h>
#include <share.h>

FILE * _fsopen(const char *nomearq, const *modo_acesso, int sinaliz_compart);
```

Os parâmetros *nomearq* e *modo_acesso* contêm ponteiros para strings de caracteres para o nome de arquivo e o modo de acesso desejado que você normalmente usaria com *fopen*. O parâmetro *sinaliz_compart* especifica o modo de compartilhamento. Se a função for bem-sucedida, ela retornará um ponteiro de arquivo. Se ocorrer um erro, a função retornará *NULL*. A Tabela 437 lista os valores válidos que você pode atribuir a *sinaliz_compart*.

Tabela 437 Valores válidos para o parâmetro ***modo_acesso***.

Sinalizador de Compartilhamento	Compartilhamento Permitido
SH_COMPAT	Permite o compartilhamento compatível
SH_DENYRW	Impede o acesso de leitura e gravação

Tabela 437 Valores válidos para o Parâmetro *modo_acesso*. (*Continuação*)

Sinalizador de Compartilhamento	Compartilhamento Permitido
SH_DENYWR	Impede o acesso de gravação
SH_DENYRD	Impede o acesso de leitura
SH_DENYNONE	Permite todos os acessos (leitura e gravação)
SH_DENYNO	Permite todos os acessos (leitura e gravação)

Os comandos a seguir, por exemplo, abrem o arquivo *autoexec.bat* para operações de leitura compartilhadas:

```
if ((pa = _fsopen("\\AUTOEXEC.BAT", "r", SH_DENYWR)) == NULL)
   printf("Erro ao abrir \\AUTOEXEC.BAT\n");
else
   // Comandos
```

438 CRIANDO UM ARQUIVO EXCLUSIVO EM UM DIRETÓRIO ESPECÍFICO

Várias dicas nesta seção exibem modos de seus programas criarem arquivos temporários. Se você normalmente trabalha com indicativos de arquivos, pode usar a função *creattemp*, que retorna um indicativo, como mostrado aqui:

```
#include <dos.h>

int creattemp(char *caminho, int atributo);
```

O parâmetro *caminho* especifica o nome do diretório dentro do qual você quer criar o arquivo. O nome precisa terminar com dois caracteres de barra invertida ('\\'). A função *creattemp* irá anexar o nome de arquivo à string para produzir um nome de caminho completo. O parâmetro *atributo* especifica os atributos de arquivo desejados (ou 0 para nenhum). A Tabela 438 lista as definições válidas para o parâmetro *atributo*.

Tabela 438 Definições válidas para o *parâmetro atributo*.

Constante	Descrição
FA_RDONLY	Arquivo de leitura-somente
FA_HIDDEN	Arquivo oculto
FA_SYSTEM	Arquivo de sistema

Se a função for bem-sucedida, ela retornará um indicativo de arquivo. Se ocorrer um erro, a função retornará -1. O programa a seguir, *criatemp.c*, usa a função *creattemp* para criar um arquivo exclusivo no diretório TEMP:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>

void main(void)
 {
   char caminho[64] = "C:\\TEMP\\";
   int indicat;

   if ((indicat = creattemp(caminho, 0)) == -1)
      printf("Erro ao criar o arquivo\n");
   else
    {
      printf("Caminho completo: %s\n", caminho);
      close(indicat);
    }
 }
```

CRIANDO UM NOVO ARQUIVO 439

Várias dicas nesta seção mostraram modos de criar arquivos. Em muitos casos, se você tentar criar um arquivo e o nome especificado na função já existir, a função truncará o conteúdo do arquivo. No entanto, você frequentemente poderá querer apenas criar um arquivo se um arquivo com o mesmo nome ainda não existir. Para esses casos, seus programas podem usar a função *creatnew*, como mostrado aqui:

```
#include <dos.h>

int creatnew(const char *nomecaminho, int atributo);
```

O parâmetro *nomecaminho* especifica o caminho completo do arquivo que você quer criar. O parâmetro *atributo* especifica os atributos de arquivo desejados (ou 0 para nenhum). A Tabela 439.1 lista as definições possíveis para o parâmetro *atributo*.

Tabela 439.1 As definições possíveis para o parâmetro ***atributo*** da função ***creatnew***.

Atributo	Significado
FA_RDONLY	Arquivo de leitura somente
FA_HIDDEN	Arquivo oculto
FA_SYSTEM	Arquivo do sistema

Se *creatnew* for bem-sucedida, ela retornará um indicativo de arquivo. Se ocorrer um erro, a função retornará o valor -1 e definirá a variável global *errno* com um dos valores listados na Tabela 439.2.

Tabela 439.2 Os valores de retorno de erro de ***creatnew***.

Erro	Significado
EXISTS	O arquivo já existe
ENOENT	Caminho não-encontrado
EMFILE	Arquivos demais abertos
EACCES	Violação de acesso

O programa a seguir, *crianovo.c*, usa a função *creatnew* para criar um arquivo chamado *novo.dat* no diretório atual. Experimente este programa e tente criar o arquivo mais de uma vez, como mostrado aqui:

```
#include <stdio.h>
#include <dos.h>
#include <io.h>

void main(void)
 {
   int indicativo;

   if ((indicativo = creatnew("NOVO.DAT", 0)) == -1)
      printf("Erro ao criar NOVO.DAT\n");
   else
     {
       printf("Arquivo criado com sucesso \n");
       close(indicativo);
     }
 }
```

USANDO OS SERVIÇOS DO DOS PARA ACESSAR UM ARQUIVO 440

Como foi visto, quando seus programas precisam acessar mais de 20 arquivos, você pode querer usar os serviços do DOS, que lhe permitem ignorar as rotinas da biblioteca de execução de C. O programa a seguir, *copiados.c*, usa os serviços do DOS para copiar o conteúdo do primeiro arquivo especificado na linha de comando para o segundo:

```
#include <stdio.h>
```

```
#include <dos.h>

void main(int argc, char **argv)
 {
   union REGS ent_regs, sai_regs;
   struct SREGS segs;
   char buffer[256];
   unsigned origem_indic, alvo_indic;

   if (*argv[1] && *argv[2])
   {
     // Abre o arquivo para cópia
     ent_regs.h.ah = 0x3D;
     ent_regs.h.al = 0;  // Abre para acesso de leitura
     ent_regs.x.dx = (unsigned) argv[1];
     segread (&segs);
     intdosx(&ent_regs, &sai_regs, &segs);
     if (sai_regs.x.cflag)
       printf("Erro ao abrir o arquivo de origem %s\n", argv[1]);
     else
       {
         origem_indic = sai_regs.x.ax;
         // Cria o arquivo-alvo, truncando um
         // arquivo existente com o mesmo nome
         ent_regs.h.ah = 0x3C;
         ent_regs.x.cx = 0; // Abre com o atributo normal
         ent_regs.x.dx = (unsigned) argv[2];
         intdosx (&ent_regs, &sai_regs, &segs);
         if (sai_regs.x.cflag)
            printf ("Erro ao criar o arquivo-alvo %s\n", argv[2]);
         else
          {
            alvo_indic = sai_regs.x.ax;
            do {
               // Lê os dados de origem
               ent_regs.h.ah = 0x3F;
               ent_regs.x.bx = origem_indic;
               ent_regs.x.cx = sizeof(buffer);
               ent_regs.x.dx = (unsigned) buffer;
               intdosx (&ent_regs, &sai_regs, &segs);
               if (sai_regs.x.cflag)
                {
                  printf("Erro ao ler o arquivo de origem\n");
                  break;
                }
               else if(sai_regs.x.ax) // Não é fim de arquivo
                {
                  // Grava os dados
                  ent_regs.h.ah = 0x40;
                  ent_regs.x.bx = alvo_indic;
                  ent_regs.x.cx = sai_regs.x.ax;
                  ent_regs.x.dx = (unsigned) buffer;
                  intdosx (&ent_regs, &sai_regs, &segs);
                  if (sai_regs.x.cflag)
                   {
                    printf("Erro ao gravar o arquivo-alvo\n");
                    break;
                   }
```

```
                }
            } while (sai_regs.x.ax != 0);
            // Fecha os arquivos
            ent_regs.h.ah = 0x3E;
            ent_regs.x.bx = origem_indic;
            intdos (&ent_regs, &sai_regs);
            ent_regs.x.bx = alvo_indic;
            intdos (&ent_regs, &sai_regs);
          }
        }
    }
    else
      printf('Especifique os arquivos de origem e de destino \n");
}
```

Nota: Como foi visto, você geralmente usará a API do Windows para efetuar atividades equivalentes de serviços do DOS no Windows. A Dica 1471 explicará como copiar arquivos usando a API do Windows.

FORÇANDO A ABERTURA DE UM ARQUIVO NO MODO BINÁRIO OU TEXTO 441

Você aprendeu anteriormente que muitos compiladores C usam a variável global *_fmode* para determinar se o programa abriu arquivos no modo texto ou binário. Ao usar a função *fopen*, você pode controlar qual modo *fopen* usar colocando a letra *t* ou *b* imediamente após o modo desejado, como mostrado na Tabela 441.

Tabela 411 Especificadores do modo do arquivo para ***fopen***.

Especificador de Acesso	Modo de Acesso
ab	Anexação, acesso no modo binário
at	Anexação, acesso no modo texto
rb	Leitura, acesso no modo binário
rt	Leitura, acesso no modo texto
wb	Gravação, acesso no modo binário
wt	Gravação, acesso no modo texto

O comando *fopen* a seguir, por exemplo, abre o arquivo *nomearq.ext* para acesso de leitura no modo binário:

```
if ((pa = fopen("NOMEARQ.EXT", "rb")))
```

LENDO LINHAS DE TEXTO 442

Quando seus programas lêem arquivos de texto, normalmente fazem isso uma linha de cada vez. Para ler uma linha de um arquivo, seus programas podem usar a função *fgets*, cujo formato é mostrado aqui:

```
#include <stdio.h>

char *fgets(char string, int limite, FILE *canal);
```

O parâmetro *string* é o buffer de caracteres no qual *fgets* lê os dados do arquivo. Normalmente seus programas irão declarar uma matriz de 128 ou 256 bytes para conter os dados. O parâmetro *limite* especifica o número de caracteres que o buffer pode conter. Quando *fgets* lê caracteres do arquivo, *fgets* lerá até *limite–1* (limite menos um) ou até o primero caractere de *nova linha* (\n), o que vier primeiro. A função então colocará um caractere *NULL* no buffer para indicar o final da string.

Muitos programas usam a função *sizeof* para especificar o tamanho do buffer, tal como *sizeof(string)*. Finalmente, o parâmetro *canal* especifica o arquivo a partir do qual *fgets* precisa ler a string. Você precisa ter aberto previamente o canal usando *fopen* ou usar um indicativo predefinido, tal como *stdin*. Se *fgets* ler informações com sucesso do arquivo, *fgets* retornará um ponteiro para a string. Se ocorrer um erro ou se chegar ao final do arquivo, *fgets* retornará *NULL*.

443 GRAVANDO LINHAS DE TEXTO

Você aprendeu na dica anterior que seus programas tipicamente irão ler uma linha de cada vez de um arquivo. Ao gravar em um arquivo, seus programas gravarão uma linha de cada vez. Para gravar uma string em um arquivo, seus programas podem usar a função *fputs*, como mostrado aqui:

```
#include <stdio.h>

int fputs(const char *string, FILE *canal);
```

A função *fputs* grava os caracteres em uma string especificada até o caractere *NULL* de finalização. Se *fputs* gravar a string com sucesso, ela retornará um valor positivo para a função chamadora. Se ocorrer um erro, *fputs* retornará a constante EOF.

444 COLOCANDO FGETS E FPUTS EM USO

Nas Dicas 442 e 443 você aprendeu que seus programas podem usar as funções *fgets* e *fputs* para ler e gravar dados em arquivos. O programa a seguir, *copiatxt.c*, usa *fgets* e *fputs* para copiar o conteúdo do primeiro arquivo especificado na linha de comando para o segundo arquivo:

```
#include <stdio.h>

void main(int argc, char **argv)
 {
   FILE *entrada, *saida;
   char string[256];

   if ((entrada = fopen(argv[1], "r")) == NULL)
      printf("Erro ao abrir %s\n", argv[1]);
   else if ((saida = fopen(argv[2], "w")) == NULL)
    {
       printf("Erro ao abrir %s\n", argv[2]);
       fclose(entrada);
    }
   else
    {
      while (fgets(string, sizeof(string), entrada))
         fputs(string, saida);
      fclose(entrada);
      fclose(saida);
    }
 }
```

Como você pode ver, o programa abre um arquivo de entrada e um arquivo de saída e depois lê e grava o texto até que a função *fgets* encontre o final do arquivo (*fgets* retorne *NULL*). Por exemplo, para copiar o conteúdo do arquivo *teste.dat* para *teste.sav*, você poderia usar o programa *copiatxt.c* como segue:

```
C:\> COPIATXT  TESTE.DAT  TESTE.SAV   <Enter>
```

445 FORÇANDO A TRADUÇÃO DE ARQUIVO BINÁRIO

Como você aprendeu, muitos compiladores usam a variável global *_fmode* para determinar o acesso de arquivo de texto ou arquivo binário. No modo texto, as funções da biblioteca de execução de C traduzem os caracteres de alimentação de linha em combinações de retorno do carro e alimentação de linha e vice-versa. Como você aprendeu, definindo a variável *_fmode* como *O_TEXT* ou *O_BINARY*, é possível controlar o modo de acesso. Além disso, colocando um *t* ou *b* dentro do modo de acesso especificado em *fopen*, você pode definir o modo de acesso para o modo de acesso binário ou texto. Por exemplo, a seguinte chamada da função *fopen*, abre o arquivo *nomearq.c* para acesso de leitura no modo binário:

```
if ((pa = fopen("NOMEARQ.EXT", "rb")) == NULL)
```

COMPREENDENDO POR QUE O PROGRAMA COPIATXT NÃO PODE COPIAR ARQUIVOS BINÁRIOS 446

A Dica 444 apresentou o programa *copiatxt.c*, que copia o conteúdo do primeiro arquivo especificado na linha de comando para o segundo arquivo. Se você tentar usar *copiatxt* para copiar um arquivo binário, tal como um arquivo *exe*, a operação de cópia falhará. Quando a função *fgets* lê um arquivo de texto, *fgets* considera o caractere Ctrl+Z (o caractere ASCII 26) como o final do arquivo. Como um arquivo binário provavelmente conterá uma ou mais ocorrências do valor 26, *fgets* terminará sua operação de cópia na primeira ocorrência. Se você quiser copiar um arquivo executável ou outro arquivo binário, precisará usar as rotinas de E/S de baixo nível de C.

TESTANDO O FINAL DO ARQUIVO 447

Como você aprendeu, quando a função *fgets* encontra o final de um arquivo, ela retorna *NULL*. Da mesma forma, quando *fgetc* atinge o final de um arquivo, ela retorna EOF. Algumas vezes seus programas precisam determinar se o ponteiro de um arquivo está no final do arquivo antes de efetuar uma operação específica. Nesses casos, seus programas podem chamar a função *feof*, como mostrado aqui:

```
#include <stdio.h>

int feof(FILE *canal);
```

Se o ponteiro de arquivo especificado estiver no final do arquivo, *feof* retornará um valor diferente de zero (verdadeiro). Se ainda não tiver atingido o final do arquivo, *feof* retornará 0 (falso). O laço a seguir lê e exibe os caracteres do arquivo que correspondem ao ponteiro de arquivo *entrada*:

```
while (! feof(entrada))
    fputc(fgetc(entrada), stdout);
```

Nota: Após uma função, tal como fgetc, definir o indicador de final de arquivo para um arquivo, ele permanece definido até que o programa feche o arquivo ou chame a função rewind.

DEVOLVENDO UM CARACTERE 448

Certos programas, tais como um compilador, por exemplo, frequentemente lêem caracteres de um arquivo um de cada vez até encontrar um caractere específico (um delimitador ou símbolo especial, também chamado *token*). Após o programa encontrar o caractere, ele precisará executar um processamento específico. Após o programa completar o processamento, ele continuará a ler o arquivo. Dependendo da estrutura do arquivo que seu programa estiver lendo, algumas vezes você desejará que o programa "devolva" um caractere. Nesses casos, o programa pode usar a função *ungetc*, cujo formato é mostrado aqui:

```
#include <stdio.h>

int ungetc(int caractere, FILE *canal);
```

A função *ungetc* coloca o caractere especificado de volta no buffer do arquivo. Você somente pode "devolver" um caractere. Se você chamar *ungetc* duas vezes seguidas, o segundo caractere sobrescreverá o primeiro caractere que você devolveu. A função *ungetc* coloca o caractere especificado no membro *hold* da estrutura *FILE*. Por sua vez, a próxima operação de arquivo incluirá o caractere.

LENDO DADOS FORMATADOS DE ARQUIVO 449

Você aprendeu como usar a função *fprintf* para gravar saída formatada em um arquivo. De uma forma similar, a função *fscanf* lhe permite ler dados de arquivos formatados, exatamente como a função *scanf*, que você viu anteriormente, lhe permite ler dados formatados do teclado. O formato da função *fscanf* é como segue:

```
#include <stdio.h>

int fscanf(FILE *canal, const char *fmt[, ender_var,...]);
```

O parâmetro *canal* é um ponteiro para o arquivo a partir do qual você quer que *fscanf* leia. O parâmetro *fmt* especifica o formato de dados usando o mesmo caractere de controle que *scanf*. Finalmente, o parâmetro *ender_var* especifica um endereço no qual você quer os dados lidos. As reticências (...) que aparecem após o parâmetro *ender_var* indicam que você pode ter múltiplos endereços separados por vírgulas.

Quando termina, *fscanf* retorna o número de campos que leu. Se *fscanf* encontrar o final do arquivo, retornará a constante *EOF*. O programa a seguir, *fscanf.c*, abre o arquivo *dados.dat* para saída, grava saída formatada no arquivo usando *fprintf*, fecha o arquivo e depois o reabre para entrada, lendo seu conteúdo com *fscanf*:

```
#include <stdio.h>

void main(void)
 {
   FILE *pa;

   int idade;
   float salario;
   char nome[64];

   if ((pa = fopen("DADOS.DAT", "w")) == NULL)
     printf("Erro ao abrir DADOS.DAT para saída\n");
   else
     {

       fprintf(pa, "33 35000.0 Kris");
       fclose(pa);
       if ((pa = fopen("DADOS.DAT", "r")) == NULL)
         printf("Erro ao abrir DADOS.DAT para entrada\n");
       else
        {
          fscanf(pa, "%d %f %s", &idade, &salario, nome);
          printf("Idade %d Salário %f Nome %s\n", idade, salario, nome);
          fclose(pa);
        }
     }
 }
```

450 Posicionamento do Ponteiro de Arquivo com Base em Sua Posição Atual

Você aprendeu que o ponteiro de arquivo contém um ponteiro de posição para controlar sua posição atual dentro do arquivo. Quando você conhece o formato do seu arquivo, algumas vezes quer avançar o ponteiro de posição para um local específico antes de começar a ler o arquivo. Por exemplo, os primeiros 256 bytes do seu arquivo podem conter informações de cabeçalho que você não quer ler. Nesses casos, seus programas podem usar a função *fseek* para posicionar o ponteiro de arquivo, como mostrado aqui:

```
#include <stdio.h>

int fseek(FILE *canal, long desloc, int relativo_a);
```

O parâmetro *canal* especifica o ponteiro de arquivo que você quer posicionar. Os parâmetros *desloc* e *relativo_a* combinam para especificar a posição desejada. O *desloc* contém o byte de deslocamento no arquivo. O parâmetro *relativo_a* especifica a posição no arquivo a partir do qual *fseek* deve aplicar o deslocamento. A Tabela 450 especifica os valores que você pode usar para o parâmetro *relativo_a*.

Tabela 450 Posições no arquivo a partir das quais *fseek* pode aplicar um deslocamento

Constante	Significado
SEEK_CUR	Da posição atual no arquivo
SEEK_SET	Do início do arquivo
SEEK_END	Do final do arquivo

Para posicionar o ponteiro de arquivo imediatamente após os primeiros 256 bytes de informações de cabeçalho em um arquivo, você poderia usar *fseek*, como segue:

```
fseek(pa, 256, SEEK_SET);  // O deslocamento 0 é o início
```

Se tiver sucesso, *fseek* retornará o valor 0. Se ocorrer um erro, a função retornará um valor diferente de 0.

OBTENDO INFORMAÇÕES DO INDICATIVO DE ARQUIVO 451

Quando você trabalha com um indicativo de arquivo, algumas vezes precisa conhecer detalhes específicos sobre o arquivo correspondente, tal como a unidade de disco onde ele está armazenado. Nesses casos, seus programas podem usar a função *fstat*, que tem o seguinte formato:

```
#include <sys\stat.h>

int fstat(int indicativo, struct stat *buffer);
```

A função atribui informações específicas sobre o arquivo em uma estrutura do tipo *stat* definida dentro do arquivo de inclusão *stat.h*, como mostrado aqui:

```
struct  stat
 {
  short st_dev;     // Número da unidade de disco
  short st_ino;     // Não usado pelo DOS
  short st_mode;    // Modo de abertura do arquivo
  short st_nlink;   // Sempre 1
  short st_uid;     // Ident. do usuário - Não usado
  short st_gid;     // Ident. do grupo - Não usado
  short st_rdev;    // O mesmo que st_dev
  long st_size;     // Tamanho do arquivo em bytes
  long st_atime;    // Hora em que foi aberto pela última vez
  long st_mtime;    // O mesmo que st atime
  long st_ctime;    // O mesmo que st_atime
 };
```

Se *fstat* tiver sucesso, ela retornará o valor 0. Se ocorrer um erro, *fstat* retornará o valor -1, e definirá a variável global *errno* como *EBADF* (número de indicativo inválido). O programa a seguir, *autoinfo.c*, usa a função *fstat* para exibir a data e a hora da última modificação no arquivo *autoexec.bat*, bem como o tamanho dele:

```
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <time.h>

void main(void)
 {
   int indic;
   struct stat buffer;

   if ((indic = open("\\AUTOEXEC.BAT", O_RDONLY)) == -1)
     printf("Erro ao abrir \\AUTOEXEC.BAT\n");
   else
     {
       if (fstat(indic, &buffer))
         printf('Erro ao obter informações do arquivo\n");
       else
         {
           printf("AUTOEXEC.BAT tem %ld bytes\n",
                buffer.st_size);
           printf("Usado pela última vez em %s\n",
                ctime(&buffer.st_atime));
         }
```

```
        close(indic);
      }
}
```

Nota: A Dica 1465 detalhará como é possível obter a data e a hora de um arquivo dentro do Windows.

452 REABRINDO UM CANAL DE ARQUIVO

À medida que seus programas trabalharem com arquivos, algumas vezes você poderá querer sobrescrever um ponteiro de arquivo aberto. Por exemplo, o DOS não fornece um modo de redirecionar a saída do indicativo de arquivo *stderr* da linha de comando. No entanto, dentro do seu programa, você poderá sobrescrever o destino do ponteiro de arquivo *stderr* reabrindo-o usando a função *freopen*.

```
#include <stdio.h>

FILE *freopen(const char *nomearq, const char modo_acesso, FILE *canal);
```

A função *fropen* é similar a *fopen*, exceto que você passe para a função um ponteiro de arquivo cujo valor quer sobrescrever. Se a função for bem-sucedida, ela retornará um ponteiro para o canal de arquivo original. Se um erro ocorrer, *fropen* retornará *NULL*. O programa a seguir, *nostderr.c*, por exemplo, redireciona as funções *stderr* para o arquivo *normal.err*, em vez de para a tela:

```
#include <stdio.h>

void main(void)
 {
   if (freopen("NORMAL.ERR", "w", stderr))
      fputs("stderr foi redirecionada", stderr);
   else
      printf("Erro ao reabrir\n");
 }
```

453 COMPREENDENDO AS MATRIZES

Como você aprendeu, um tipo descreve o conjunto de valores que contém e define as operações que seus programas podem executar na variável. Exceto para strings de caracteres, todos os tipos que você examinou até aqui podem conter somente um valor. À medida que seus programas começarem a realizar trabalho mais útil, algumas vezes você desejará que uma variável contenha muitos valores. Por exemplo, a variável *nota* pode controlar as notas obtidas por 100 alunos no exame. Da mesma forma, a variável *salarios* poderia controlar o salário de cada funcionário da companhia. Uma matriz é uma estrutura de dados que pode armazenar múltiplos valores do mesmo tipo. Por exemplo, você pode criar uma matriz que possa conter 100 valores do tipo *int* e uma segunda matriz que possa conter 25 valores do tipo *float*.

Todo valor que você atribui a uma matriz precisa ser do mesmo tipo que o tipo da matriz. Nesta seção você aprenderá como criar e trabalhar com matrizes em seus programas. Após você trabalhar com uma ou duas matrizes, verá que elas são fáceis de entender. Se você já se sente à vontade quando trabalha com strings, logo se sentirá igualmente à vontade ao trabalhar com matrizes. Lembre-se, uma string de caracteres é simplesmente uma matriz de caracteres.

454 DECLARANDO UMA MATRIZ

Na Dica 453 você aprendeu que uma *matriz* é uma variável que pode armazenar múltiplos valores do mesmo tipo. Para declarar uma matriz, você precisa especificar o tipo desejado (tal como *int*, *float*, ou *double*), bem como o tamanho da matriz. Para especificar o tamanho de uma matriz, coloque o número de valores que a matriz pode armazenar dentro de colchetes após o nome da matriz. Por exemplo, a declaração a seguir cria uma matriz chamada *notas*, que pode armazenar 100 notas de exame do tipo *int*:

```
int notas[100];
```

De forma similar, a seguinte declaração cria uma matriz do tipo *float*, que contém 50 salários:

```
float salarios[50];
```

Quando você declara uma matriz, C aloca memória suficiente para conter todos os elementos. O primeiro item está na posição 0. Por exemplo, nas matrizes *notas* e *salarios*, os comandos a seguir atribuem os valores 80 e 35000 aos primeiros elementos da matriz:

```
notas[0] = 80;
salarios[0] = 35000;
```

Como o primeiro elemento da matriz inicia no deslocamento 0, o último elemento da matriz ocorre uma posição antes do tamanho da matriz. Dadas as matrizes anteriores, *notas* e *salarios*, os comandos a seguir atribuem valores ao último elemento de cada matriz:

```
notas[99] = 75;
salarios[49] = 240000;
```

VISUALIZANDO UMA MATRIZ 455

Como foi visto, uma matriz é uma variável que pode armazenar múltiplos valores do mesmo tipo. Para lhe ajudar a compreender melhor como uma matriz armazena informações, considere a seguinte declaração de matrizes:

```
char string[64];
float salarios[50];
int notas[100];
long planetas[13];
```

Após você atribuir valores a cada matriz, as matrizes irão residir na memória de uma maneira similar à mostrada na Figura 455.

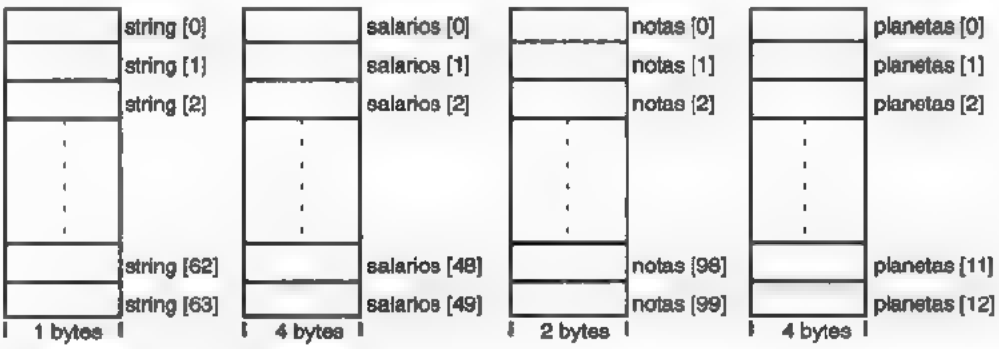

Figura 455 Armazenando valores em matrizes.

Como você pode ver, o primeiro valor de cada matriz reside no deslocamento 0. Na seção Introdução à Linguagem C, anteriormente, uma variável é um nome que você atribui a uma ou mais posições de memória. Em uma matriz, você pode ter um grande número de posições de memória que correspondem a uma única matriz.

COMPREENDENDO OS REQUISITOS DE ARMAZENAMENTO DE UMA MATRIZ 456

Como você aprendeu, uma matriz é uma coleção nomeada de valores do mesmo tipo. Ao declarar uma matriz, o compilador C aloca memória suficiente para conter o número de valores que você especifica. A quantidade real de memória que o compilador aloca depende do tipo da matriz. Por exemplo, uma matriz de 100 elementos do tipo *int* normalmente irá requerer 100* 2 ou 200 bytes de memória. Por outro lado, uma matriz de 100 elementos do tipo *float* irá requer 100* 4 bytes ou 400 bytes O programa a seguir, *tamatriz.c*, usa o operador *sizeof* para exibir a quantidade de memória que os diferentes tipos de matriz requerem:

```
#include <stdio.h>

void main(void)
```

```
{
  int notas[100];
  float salar[100];
  char string[100];

  printf("Memória para conter int notas[100] %d bytes\n", sizeof(notas));
  printf("Memória para conter float salar[100] %d bytes\n",
         sizeof(salar));
  printf("Memória para conter char string[100] %d bytes\n",
         sizeof(string));
}
```

Quando você compilar e executar o programa *tamatriz.c*, sua tela exibirá o seguinte resultado:

```
Memória para conter int notas[100]  200 bytes
Memória para conter float salar[100]  400 bytes
Memória para conter char string[100]  100 bytes
C:\>
```

457 INICIALIZANDO UMA MATRIZ

Neste livro, muitos programas inicializaram as strings de caracteres como segue:

```
char titulo[] = "Bíblia do Programador C/C++";
char secao[64] = "Matrizes";
```

No primeiro caso, o compilador C alocará 28 bytes para armazenar a string. No segundo caso, o compilador alocará uma matriz de 64 bytes, inicializando os primeiros oito caracteres com as letras "Matrizes" e o caractere *NULL*. A maioria dos compiladores também inicializará as posições de byte restantes com *NULL*. Quando você declara matrizes de outros tipos, pode inicializar as matrizes da mesma forma. Por exemplo, o comando a seguir inicializa a matriz de inteiros *notas* com os valores 80, 70, 90, 85 e 80:

```
int notas[5] = {80, 70, 90, 85, 80};
```

Quando você atribui valores iniciais a uma matriz, precisa delimitar os valores por abre e fecha chaves ({}). No caso anterior, o tamanho da matriz é igual ao número de valores atribuídos. O comando a seguir, no entanto, atribui quatro valores de *ponto flutuante* a uma matriz que pode armazenar 64 valores:

```
float salar[64] = { 25000.0, 32000.0, 44000.0, 23000.0 };
```

Dependendo do seu compilador, ele pode atribuir 0 aos elementos aos quais seu programa não atribuiu valores explícitos. No entanto, como regra, você não deve assumir que o compilador inicializará os outros elementos. Além disso, se você não especificar um tamanho de matriz, o compilador alocará memória suficiente para conter somente os valores que você especificar. Por exemplo, a seguinte declaração de matriz cria uma matriz grande o suficiente para conter três valores do tipo *long*:

```
long planetas[] = { 1234567L, 654321L, 1221311L };
```

458 ACESSANDO ELEMENTOS DA MATRIZ

Os valores armazenados em uma matriz são chamados *elementos da matriz*. Para acessar um elemento da matriz, você especifica o nome da matriz e o elemento que deseja. O programa a seguir, *elemento.c*, inicializa a matriz *notas* e depois usa *printf* para exibir os valores dos elementos.

```
#include <stdio.h>

void main(void)
{
  int notas[5] = {80, 70, 90, 85, 80};

  printf("Valores da Matriz\n");
  printf("notas[0] %d\n", notas[0]);
```

```
    printf("notas[1] %d\n", notas[1]);
    printf("notas[2] %d\n", notas[2]);
    printf("notas[3] %d\n", notas[3]);
    printf("notas[4] %d\n", notas[4]);
  }
```

Quando você compilar e executar o programa *elemento.c*, sua tela exibirá o seguinte resultado:

```
Valores da Matriz
notas[0] = 80
notas[1] = 70
notas[2] = 90
notas[3] = 85
notas[4] = 80
C:\>
```

Como você pode ver, para acessar um determinado elemento da matriz, você especifica o número do elemento que quer dentro de colchetes após o nome da matriz.

PERCORRENDO EM UM LAÇO OS ELEMENTOS DA MATRIZ 459

Na dica anterior você usou os valores de 0 até 4 para exibir os elementos da matriz *notas*. Quando você referencia muitos elementos da matriz, especificar números para cada elemento da matriz individualmente pode ser demorado e tedioso. Como uma alternativa, seus programas podem usar uma variável para referenciar os elementos da matriz. Por exemplo, assumindo que a variável *i* contenha o valor 2, o comando a seguir atribuiria o valor 80 a *matriz[2]*:

```
i = 2;

matriz[i] = 80;
```

O programa a seguir, *exibemat.c*, usa a variável *i* e um laço *for* para exibir os elementos da matriz *notas*.

```
#include <stdio.h>

void main(void)
 {
   int notas[5] = {80, 70, 90, 85, 80};
   int i;

   printf("Valores da Matriz\n");
   for (i = 0; i < 5; i++)
     printf("notas[%d] %d\n", i, notas[i]);
 }
```

USANDO CONSTANTES PARA DEFINIR AS MATRIZES 460

Como você aprendeu, quando seus programas trabalham com matrizes, você precisa especificar o tamanho da matriz. Por exemplo, o programa a seguir, *5_valor.c*, declara uma matriz de cinco valores, e, depois, usa um laço *for* para exibir os valores da matriz:

```
#include <stdio.h>

void main(void)
 {
   int valores[5] = {80, 70, 90, 85, 80};
   int i;

   for (i = 0; i < 5; i++)
     printf("valores[%d] %d\n", i, valores[i]);
 }
```

Por exemplo, assuma que você queira, mais tarde, alterar o programa *5_valor.c* para que ele suporte 10 valores. Você precisará então alterar não somente a declaração da matriz, mas também o laço *for*. Quanto mais alterações você fizer em um programa, maiores as chances de errar. Como uma alternativa, seus programas deverão declarar matrizes usando constantes. O programa a seguir, *5_const.c*, declara uma matriz com base na constante *TAM_MATRIZ*. Como você pode ver, o programa não somente usa a constante para declarar a matriz, mas também usa a constante como a condição final para o laço *for*:

```
#include <stdio.h>

#define TAM_MATRIZ 5

void main(void)
 {
   int valores[TAM_MATRIZ] = {80, 70, 90, 85, 80};
   int i;

   for (i = 0; i < TAM_MATRIZ; i++)
     printf("valores[%d] %d\n", i, valores[i]);
 }
```

Se você, mais tarde, alterar o tamanho da matriz, poderá alterar o valor atribuído à constante *TAM_MATRIZ* para que o programa automaticamente atualize os laços que controlam a matriz como o tamanho da matriz.

461 PASSANDO UMA MATRIZ A UMA FUNÇÃO

Como você aprendeu, uma matriz é uma variável que pode armazenar múltiplos valores do mesmo tipo. Como todas as variáveis, seus programas podem passar matrizes para as funções. Quando você declara uma função que trabalha com um parâmetro matriz, precisa informar o compilador. Por exemplo, o seguinte programa, *matriz-fu.c*, usa a função *exibe_matriz* para exibir os valores em uma matriz. Como você pode ver, seu programa passa para a função tanto a matriz como o número de elementos que a matriz contém, como mostrado aqui:

```
#include <stdio.h>

void exibe_matriz(int valores[], int num_de_elementos)
 {
   int i;

   for (i = 0; i < num_de_elementos; i++)
    printf("%d\n", valores[i]);
 }

void main(void)
 {
   int notas[5] = {70, 80, 90, 100, 90};

   exibe_matriz(notas, 5);
 }
```

Quando uma função recebe uma matriz como parâmetro, seu programa não precisa especificar o tamanho da matriz na declaração do parâmetro. No caso da função *exibe_valores*, os colchetes após o nome da variável *valor* informam o compilador de que o parâmetro é uma matriz. Sabendo que o parâmetro é uma matriz, o compilador não se preocupa com o tamanho da matriz que seu programa passa para a função.

462 REVISITANDO AS MATRIZES COMO FUNÇÕES

Na dica anterior você viu que, ao declarar o parâmetro formal para uma matriz, você não precisa declarar o tamanho da matriz. Em vez disso, você pode especificar somente o abre e fecha colchetes. O programa a seguir, *paramatr.c*, passa três matrizes diferentes (de diferentes tamanhos) para a função *exibe_valores*:

```
#include <stdio.h>
```

```
void exibe_matriz(int valores[], int num_de_elementos)
 {
   int i;

   printf("Prestes a exibir %d valores\n", num_de_elementos);
   for (i = 0; i < num_de_elementos; i++)
      printf("%d\n", valores[i]);
 }

void main(void)
 {
   int notas[5] = {70, 80, 90, 100, 90};
   int conta[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int pequeno[2] = {-33, -44};

   exibe_matriz(notas, 5);
   exibe_matriz(conta, 10);
   exibe_matriz(pequeno, 2);
 }
```

Quando você compilar e executar o programa *paramatr.c*, sua tela exibirá os valores de cada matriz. Como você aprendeu, a função não se preocupa com o tamanho da matriz. No entanto, observe que as matrizes que o programa *paramatr.c* passa para a função são todas do tipo *int*. Se você tentasse passar uma matriz do tipo *float* para a função, o compilador geraria um erro.

COMPREENDENDO COMO AS MATRIZES DE STRING DIFEREM 463

Muitas dicas apresentadas neste livro passaram strings para funções. Em muitos casos, as funções não especificaram o tamanho da string. Por exemplo, o comando a seguir usa a função *strupr* para converter uma string para maiúsculas:

```
char titulo[64] = "Bíblia do Programador C/C++, do Jamsa!";

strupr(titulo);
```

Como você aprendeu, em C, o caractere *NULL* representa o final de uma string de caracteres. Portanto, as funções podem procurar o caractere *NULL* nos elementos da matriz para determinar onde a matriz termina. No entanto, as matrizes de outros tipos, tais como *int*, *float* ou *long*, não têm um caractere finalizador equivalente. Portanto, você normalmente precisa passar para as funções que trabalham com matrizes o número de elementos que a matriz contém.

PASSANDO MATRIZES PARA A PILHA 464

Várias dicas anteriores discutiram a passagem de matrizes como parâmetros para as funções. Quando você passa uma matriz para uma função, C coloca o endereço do primeiro elemento da matriz na pilha. Por exemplo, a Figura 464 ilustra a matriz *notas* e uma chamada de função *exibe_matriz* usando *notas*.

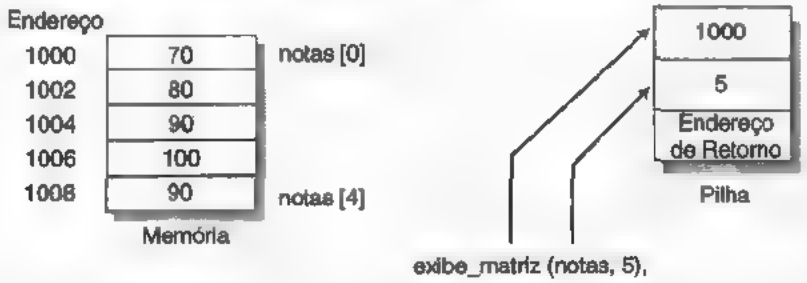

Figura 464 Quando você passa um parâmetro matriz, C coloca o endereço inicial da matriz na pilha.

Como você pode ver, C coloca somente o endereço inicial da matriz na pilha. Além disso, observe que a função não recebe nenhuma informação de C com relação ao tamanho da matriz.

465 Determinando Quantos Elementos uma Matriz Pode Armazenar

Já vimos que, dependendo do tipo de uma matriz, a quantidade real de memória que uma matriz pode consumir diferirá. Se você estiver trabalhando no ambiente do DOS, a quantidade de memória que suas matrizes podem consumir dependerá do modelo de memória atual. Em geral, uma matriz não pode consumir mais do que 64Kb de espaço. O programa a seguir, *mui_gran.c*, pode não passar na compilação porque as matrizes consomem muita memória:

```
void main(void)
 {
   char string[66000L];  // 66.000 bytes
   int valores[33000L];  // 33.000 * 2 = 66.000 bytes
   float numeros[17000]; // 17.000 * 4 = 68.000 bytes
 }
```

Nota: Como o Windows usa o modelo de memória virtual para gerenciar a memória, ele não coloca limites no tamanho da matriz iguais ao do DOS. Por exemplo, você poderá declarar strings (matrizes de caracteres) no Windows tão grandes quanto INT_MAX (2.147.483.647) caracteres em comprimento. No entanto, se você tentar declarar uma variável grande demais dentro de uma janela do DOS, provocará um erro na pilha e o Windows fechará a janela.

466 Usando o Modelo de Memória Huge para as Matrizes Grandes

Se a quantidade de memória que uma matriz consome exceder 64Kb, você poderá instruir muitos compiladores baseados no DOS a usar o modelo de memória *huge* tratando a matriz como um ponteiro e incluindo a palavra *huge* dentro da declaração, como mostrado aqui:

```
float huge valores[17000];
```

No programa a seguir, *huge_ftl.c*, cria uma matriz *huge* de pontos flutuantes:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
 {
   int i;
   float huge *valores;

   if ((valores = (float huge *) halloc (17000, sizeof(float))) == NULL)
      printf ("Erro ao alocar a matriz huge \n");
   else
      {
        printf("Preenchendo a matriz\n");
        for (i = 0; i < 17000; i++)
           valores[i] = i * 1.0;
        for (i = 0; i < 17000; i++)
           printf ("%8.1f ", valores[i]);
        hfree(valores);
      }
 }
```

Nota: Como o Windows usa o modelo de memória virtual para gerenciar a memória, ele não coloca limites no tamanho da matriz, como um programa do DOS. Por exemplo, você pode declarar uma matriz **unsigned char** *no Windows do tamanho INT_MAX (2.147.483.647) sem usar a palavra-chave* **huge**. *No entanto, se você tentar declarar uma variável de tamanho excessivo dentro de uma janela do DOS sem a palavra-chave* **huge**, *causará um erro na pilha, e o Windows fechará a janela.*

A BARGANHA ENTRE MATRIZES E MEMÓRIA DINÂMICA 467

À medida que você ficar mais experiente e começar a usar ponteiros dentro de C, poderá começar a usar matrizes com menos freqüência e, em seu lugar, alocar memória dinamicamente sempre que precisar de mais. Existem várias barganhas que você precisa considerar ao determinar se deve usar memória dinâmica ou uma matriz. Para começar, muitos usuários acham as matrizes mais simples de compreender e de usar. Como resultado, seu programa poderá ser mais fácil para outros programadores seguirem. Segundo, como o compilador aloca espaço para matrizes, seus programas não experimentam a sobrecarga de execução associada com a alocação dinâmica de memória. Como resultado, um programa baseado em matrizes tem uma execução ligeiramente mais rápida.

No entanto, como você aprendeu, ao declarar uma matriz, você precisará especificar seu tamanho. Se você não souber de que tamanho precisará, poderá ter uma tendência de alocar uma matriz maior que o necessário. Como resultado, poderá haver desperdício de memória. Por outro lado, se o tamanho da matriz for muito pequeno, você precisará editar o programa, alterar o tamanho da matriz e recompilar o programa.

Quando você declarar uma matriz dentro de seus programas, lembre-se de que poderá efetuar processamento idêntico alocando memória dinamicamente. Como aprenderá ainda nesta seção, mais à frente, você poderá referenciar dinamicamente a memória alocada usando índices de matrizes e eliminar a confusão de ponteiros que normalmente frustra os novos programadores C. Como a maioria dos sistemas operacionais permite que os programas aloquem memória muito rapidamente, você pode preferir a flexibilidade e as maiores oportunidades de gerenciamento de memória proporcionadas pela alocação dinâmica de memória em relação às matrizes, apesar da pequena sobrecarga imposta ao sistema.

COMPREENDENDO AS MATRIZES MULTIDIMENSIONAIS 468

Como você aprendeu, uma matriz é uma variável que pode armazenar múltiplos valores do mesmo tipo. Em todos os exemplos apresentados até aqui, as matrizes consistiram de uma única fileira de dados. No entanto, C também permite matrizes bi, tri e multidimensionais. O melhor modo de visualizar uma matriz bidimensional é com uma tabela com linhas e colunas. Se uma matriz contém três dimensões, visualize-a como várias páginas, cada uma das quais contendo uma tabela bidimensional, como mostrado na Figura 468.

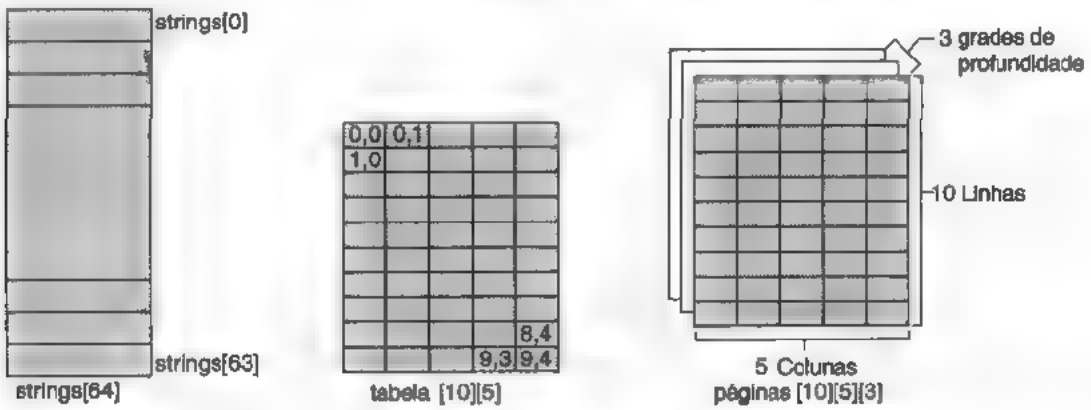

Figura 468 Modelo lógico das matrizes multidimensionais.

As seguintes declarações de matrizes criam as matrizes mostradas na Figura 468:

```
char strings[64];
int tabela[10][5];
float paginas[10][5][3];
```

469 COMPREENDENDO AS LINHAS E COLUNAS

Como você aprendeu, C suporta matrizes multidimensionais que são similares às tabelas de valores. Quando você trabalhar com uma matriz bidimensional, pense na matriz como uma tabela de linhas e colunas. As linhas da tabela vão da esquerda para a direita, enquanto as colunas vão de cima para baixo, como mostrado na Figura 469.

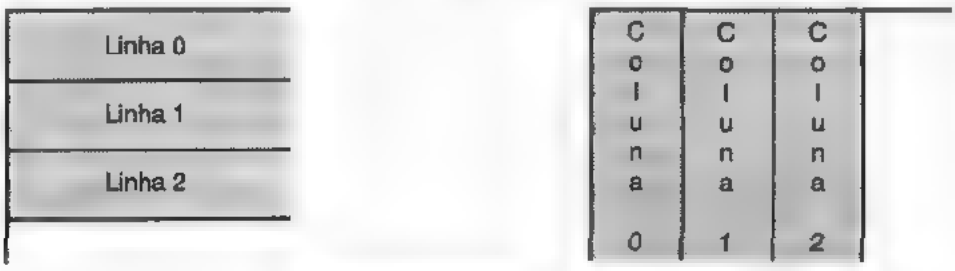

Figura 469 Linhas e colunas em uma matriz bidimensional.

Ao declarar uma matriz bidimensional, o primeiro valor que você especificar informará o número de linhas; e o segundo valor, o número de colunas:

```
int tabela [2][3];
```

470 ACESSANDO ELEMENTOS EM UMA MATRIZ BIDIMENSIONAL

Como você aprendeu, pode visualizar melhor uma matriz bidimensional como uma tabela que consiste de linhas e colunas. Para referenciar um elemento específico da matriz, você precisa especificar a posição de linha e coluna correspondente. A Figura 470 ilustra os comandos que acessam elementos específicos dentro de uma matriz *tabela*.

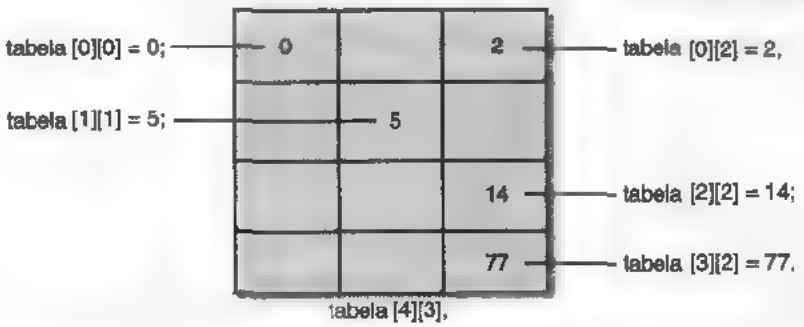

Figura 470 Para acessar elementos em uma matriz bidimensional, você precisa especificar a posição de linha e coluna do elemento.

Como pode ver, quando você acessa uma matriz bidimensional, os deslocamentos de linha e coluna iniciam em 0.

471 INICIALIZANDO ELEMENTOS EM UMA MATRIZ BIDIMENSIONAL

Na Dica 457 vimos que, para inicializar elementos da matriz, você pode colocar os valores do elemento dentro de abre e fecha colchetes após a declaração da matriz. O comando a seguir usa a mesma técnica para inicializar uma matriz bidimensional. No entanto, neste caso, o comando especifica os valores para cada linha da matriz dentro de chaves:

```
int tabela[2][3] = {{1, 2, 3},
                    {4, 5, 6}};
```

O compilador C inicializará os elementos da matriz, como mostrado na Figura 471.

Figura 471 Inicializando os elementos de uma matriz bidimensional.

De um modo similar, o comando a seguir inicializa os elementos de um matriz maior:

```
int vendas[4][5]  {{1, 2, 3, 4, 5},
                   {6, 7, 8, 9, 10},
                   {11, 12, 13, 14, 15},
                   {16, 17, 18, 19, 20}};
```

DETERMINANDO O CONSUMO DE MEMÓRIA DE UMA MATRIZ MULTIDIMENSIONAL 472

Na Dica 456 você aprendeu que seus programas podem determinar a quantidade de memória que uma matriz consome multiplicando o número de elementos na matriz pelo número de bytes requerido para representar o tipo da matriz (tal como 2 para *int*, 4 para *float* e assim por diante). Para determinar a memória que uma matriz multidimensional consome, você pode efetuar o mesmo cálculo. Para determinar o número de elementos em uma matriz multidimensional, simplesmente multiplique o número de linhas pelo número de colunas. As expressões a seguir ilustram a quantidade de memória que diferentes declarações de matriz consomem:

```
int a[5][10];      // 2 * 5 * 10 == 100 bytes
float b[5][8];     // 4 * 5 * 8  == 160 bytes
int c[3][4][5];    // 2 * 3 * 4  == 120 bytes
```

O programa a seguir, *md_taman.c*, usa o operador *sizeof* para determinar o número de bytes que as diferentes declarações de matrizes consomem:

```
#include <stdio.h>

void main(void)
 {
   int caixa[3][3];
   float vendas_ano[52][5];
   char paginas[40][60][20];

   printf("Memória para int caixa[3][3] %d bytes\n", sizeof(caixa));
   printf("Memória para float vendas_ano[52][5] %d bytes\n",
         sizeof(vendas_ano));
   printf("Memória para char paginas[40][60][20] %ld bytes\n", sizeof
(paginas));
 }
```

Quando você compilar e executar o programa *md_taman.c*, sua tela exibirá o seguinte:

```
Memória para int caixa[3][3] 18 bytes
Memória para float vendas_ano[52][5] 1040 bytes
Memória para char paginas[40][60][20] 48000 bytes
C:\>
```

PERCORRENDO EM UM LAÇO UMA MATRIZ BIDIMENSIONAL 473

Na Dica 458 você aprendeu como usar uma variável para acessar os elementos de uma matriz. Quando seus programas trabalham com matrizes bidimensionais, você normalmente usa duas variáveis para acessar elementos da matriz. O programa a seguir, *exibe_2d.c*, usa as variáveis *linha* e *coluna* para exibir os valores contidos dentro da matriz *tabela:*

```
#include <stdio.h>

void main(void)
 {
   int linha, coluna;
   float tabela[3][5] = {{1.0, 2.0, 3.0, 4.0, 5.0},
                         {6.0, 7.0, 8.0, 9.0, 10.0},
                         {11.0, 12.0, 13.0, 14.0, 15.0}};

   for (linha = 0; linha < 3; linha++)
      for (coluna = 0; coluna < 5; coluna++)
        printf("tabela[%d][%d] = %f\n", linha, coluna, tabela[linha][coluna]);
 }
```

Colocando laços *for* um dentro do outro, como mostrado, o programa exibirá os elementos contidos na primeira linha da matriz (1.0 até 5.0). Em seguida, o programa irá se mover para a próxima linha, e, depois, para a terceira linha, exibindo os elementos dentro de cada linha.

474 PERCORRENDO UMA MATRIZ TRIDIMENSIONAL

Na dica anterior vimos como percorrer uma matriz bidimensional usando duas variáveis chamadas *linha* e *coluna*. O programa a seguir, *exibe_3d.c*, usa as variáveis *linha*, *coluna* e *tabela* para percorrer uma matriz tridimensional:

```
#include <stdio.h>

void main(void)
 {
   int linha, coluna, tabela;
   float valores[2][3][5] = {
                        {{1.0, 2.0, 3.0, 4.0, 5.0},
                         {6.0, 7.0, 8.0, 9.0, 10.0},
                         {11.0, 12.0, 13.0, 14.0, 15.0}},

                         {{16.0, 17.0, 18.0, 19.0, 20.0},
                          {21.0, 22.0, 23.0, 24.0, 25.0},
                          {26.0, 27.0, 28.0, 29.0, 30.0}}
                        };

   for (linha = 0; linha < 2; linha++)
      for (coluna = 0; coluna < 3; coluna++)
         for (tabela = 0; tabela < 5; tabela++)
            printf("valores[%d][%d][%d] = %f\n", linha, coluna, tabela,
                valores[linha][coluna][tabela]);
 }
```

475 INICIALIZANDO MATRIZES MULTIDIMENSIONAIS

Na Dica 474 você aprendeu como exibir o conteúdo de uma matriz tridimensional usando três variáveis: *linha*, *coluna* e *tabela*. O programa *exibe_3d.c*, apresentado na dica anterior, inicializou os valores da matriz tridimensional, como mostrado aqui:

```
   float valores[2][3][5] = {
                        {{1.0, 2.0, 3.0, 4.0, 5.0},
                         {6.0, 7.0, 8.0, 9.0, 10.0},
                         {11.0, 12.0, 13.0, 14.0, 15.0}},

                         {{16.0, 17.0, 18.0, 19.0, 20.0},
                          {21.0, 22.0, 23.0, 24.0, 25.0},
                          {26.0, 27.0, 28.0, 29.0, 30.0}}
                        };
```

À primeira vista, inicializar uma matriz multidimensional pode parecer confuso. Para compreender melhor como inicializar essas matrizes, esta dica apresenta várias inicializações de exemplo. À medida que você examina as inicializações, faça as inicializações da direita para a esquerda.

```
int a[1][2][3] =     {
                      { {1, 2, 3}, {4, 5, 6} }
                     }  // Chaves da matriz
int b[2][3][4]=      {
                      { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} },
                      { {13, 14, 15, 16},{17, 18, 19, 20}, {21, 22, 23, 24} }
                     };   // Chaves da matriz

int c[3][2][4] =     {
                      { {1, 2, 3, 4}, {5, 6, 7, 8} },
                      { {9, 10, 11, 12},  {13, 14, 15, 16}},
                      { {17, 18, 19, 20}, {21, 22, 23, 24}}
                     };   // Chaves da matriz

int d[1][2][3][4] = {
                       {{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
                       {{13, 14, 15, 16}, {17, 18, 19, 20}, {21, 22, 23, 24}}}
                     };    // Chaves da matriz
```

Cada inicialização de matriz recebe um conjunto de chaves externas. Dentro das chaves externas, você então define os diferentes elementos da matriz dentro de chaves adicionais.

PASSANDO UMA MATRIZ BIDIMENSIONAL PARA UMA FUNÇÃO 476

À medida que seus programas forem trabalhando com matrizes multidimensionais, algumas vezes você precisará escrever funções que trabalham com matrizes. Na Dica 461, você aprendeu que, ao passar matrizes para uma função, não precisará especificar o número de elementos na matriz. Quando você trabalhar com matrizes bidimensionais, não precisará especificar o número de linhas na matriz, mas, sim, especificar o número de colunas. O programa a seguir, *func_2d.c*, usa a função *exibe_2d_matriz* para exibir o conteúdo de variáveis matrizes bidimensionais:

```
#include <stdio.h>

void exibe_2d_matriz(int matriz[][10], int linhas)
 {
   int i, j;

   for (i = 0; i < linhas; i++)
     for (j = 0; j < 10; j++)
       printf("matriz[%d][%d] = %d\n", i, j, matriz[i][j]);
 }

void main(void)
 {
  int a[1][10]={{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}};
  int b[2][10]={{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                {11, 12, 13, 14, 15, 16, 17, 18, 19, 20}};
  int c[3][10]={{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
                {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};

  exibe_2d_matriz(a, 1);
  exibe_2d_matriz(b, 2);
  exibe_2d_matriz(c, 3);
 }
```

477 TRATANDO AS MATRIZES MULTIDIMENSIONAIS COMO UMA DIMENSÃO

Na dica anterior mostramos que quando ao passar uma matriz bidimensional para uma função, e querendo acessar as posições de linha e coluna da matriz, você precisará especificar o número de colunas, como mostrado aqui:

```
void exibe_2d_matriz(int matriz[][10], int linhas)
```

Se você quiser trabalhar com os elementos de uma matriz multidimensional, mas sem precisar acessar os elementos em suas posições de linha ou coluna, suas funções poderão tratar a matriz multidimensional como se ela tivesse uma dimensão. O programa a seguir, *soma_2d.c*, retorna a soma dos valores em uma matriz bidimensional:

```
#include <stdio.h>

long soma_matriz(int matriz[], int elementos)
 {
   long soma = 0;
   int i;

   for (i = 0; i < elementos; i++)
      soma += matriz[i];
   return(soma);
 }

void main(void)
 {
  int a[10]  =   {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int b[2][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                  {11, 12, 13, 14, 15, 16, 17, 18, 19,20}};
  int c[3][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                   {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
                   {21, 22, 23, 24, 25, 26, 27, 28, 29,30}};

  printf("Soma dos elementos da primeira matriz %d\n", soma_matriz(a, 10));
  printf("Soma dos elementos da segunda matriz %d\n", soma_matriz(b, 20));
  printf("Soma dos elementos da terceira matriz %d\n", soma_matriz(c, 30));
 }
```

Como você pode ver, a função *soma_matriz* suporta matrizes de uma, duas ou múltiplas dimensões. Para compreender como *soma_matriz* funciona, você precisa primeiro compreender como C armazena matrizes multidimensionais na memória. A Dica 478, a seguir, discute como C armazena matrizes multidimensionais em detalhe.

478 COMPREENDENDO COMO C ARMAZENA MATRIZES MULTIDIMENSIONAIS

Na Dica 454 você aprendeu que, quando declara uma matriz, tal como *int notas[100]*, C aloca memória suficiente para conter cada elemento da matriz. Quando você aloca uma matriz multidimensional, o mesmo é verdadeiro. Embora as matrizes multidimensionais conceitualmente consistam de linhas, colunas e páginas, para o compilador, uma matriz multidimensional é uma seqüência longa de bytes. Por exemplo, assuma que seu programa declare a seguinte matriz:

```
int tabela[3][5];
```

A Figura 478 ilustra a aparência conceitual da matriz e o uso real da memória.

Na Dica 477 você criou uma função que tratava uma matriz multidimensional como uma dimensão para somar os valores que a matriz continha. Como o compilador C na verdade mapeia a matriz multidimensional como um intervalo de memória de uma dimensão, é válido tratar a matriz como uma dimensional.

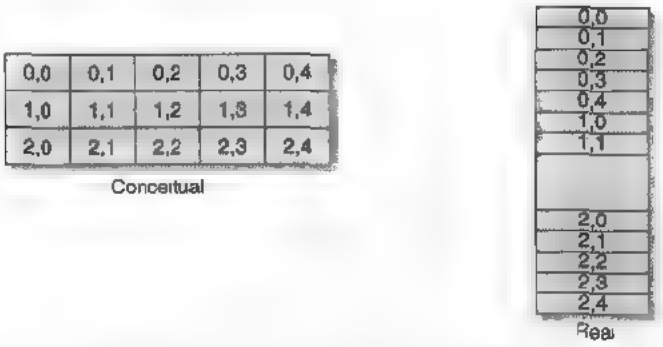

Figura 478 Mapeando uma matriz multidimensional para a memória.

COMPREENDENDO A ORDEM LINHA POR LINHA VERSUS COLUNA POR COLUNA 479

Na dica anterior vimos que o compilador C mapeia matrizes multidimensionais na memória unidimensional. Quando o compilador mapeia uma matriz multidimensional na memória, ele tem duas opções. Como mostrado na Figura 479, o compilador pode colocar os elementos da linha da matriz antes dos valores da coluna, ou o compilador pode colocar os elementos da coluna primeiro.

Figura 479 Mapeando os elementos da matriz na memória.

Quando o compilador coloca os elementos da linha da matriz na memória antes dos elementos da coluna, o compilador está efetuando a ordenação por linha. Da mesma forma, quando o compilador coloca os elementos da coluna primeiro, ele efetua a ordenação por coluna. Os compiladores C armazenam matrizes multidimensionais na ordem linha por linha.

MATRIZES DE ESTRUTURAS DE MATRIZES 480

As matrizes e estruturas lhe permitem agrupar informações relacionadas. Como você aprendeu, C lhe permite criar matrizes de estruturas ou usar matrizes como membros de estruturas. Em geral, C não coloca um limite na profundidade em que seus programas podem ir com respeito às estruturas de dados embutidas uma dentro da outra. Por exemplo, a seguinte declaração cria uma matriz de 100 estruturas de funcionários. Dentro de cada estrutura esta uma matriz de estruturas *Data* que correspondem às datas de contratação, da primeira avaliação e da última avaliação:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char ssan[11];      / Número do seguro social
   int categoria;
   float salario;
   unsigned num_funcionario;
   struct Data
   {
      int dia;
      int mes;
      int ano;
```

```
    } emp_datas[3];
} equipe[100];
```

Para acessar membros e elementos da matriz, você simplesmente trabalha da esquerda para a direita, iniciando de fora e indo para dentro. Por exemplo, os seguintes comandos atribuem a data de contratação de um funcionário:

```
equipe[10].emp_datas[0].dia = 17;
equipe[10].emp_datas[0].mes = 7;
equipe[10].emp_datas[0].ano = 1998
```

Embora embutir estruturas e matrizes uma dentro da outra possa ser muito conveniente, lembre-se de que quanto mais níveis seus programas usarem, mais difícil a estrutura se tornará para outros programadores compreenderem.

481 COMPREENDENDO AS UNIÕES

Como você aprendeu, as estruturas permitem que seus programas armazenem informações relacionadas. Dependendo do propósito do seu programa, algumas vezes as informações que você armazena em uma estrutura serão apenas um dentre dois valores. Por exemplo, assuma que seu programa controle dois valores especiais de data para cada funcionário. Para os funcionários atuais, o programa controla o número de dias trabalhados. Para um funcionário que tenha sido desligado da companhia, o programa controla a última data em que o funcionário trabalhou. Um modo de controlar essas informações é usar uma estrutura, como mostrado aqui.

```
struct EmpDatas
 {
   int dias_trabalhados;
   struct UltData
   {
      int dia;
      int mes;
      int ano;
   } ultimo_dia;
 };
```

Como o programa usará os membros *dias_trabalhados* ou *ultimo_dia*, a memória que contém o valor não-usado para cada funcionário será desperdiçada. Como uma alternativa, C permite que seus programas usem uma união, que aloca somente a memória que o maior membro da união requer, como mostrado aqui:

```
union EmpDatas
 {
   int dias_trabalhados;
   struct UltData
   {
      int dia;
      int mes;
      int ano;
   } ultimo_dia;
 };
```

Para acessar os membros da união, você usa o operador *ponto* (.) exatamente como faria com uma estrutura. No entanto, ao contrário da estrutura, a união somente pode armazenar o valor de um membro. A Figura 481 ilustra como C aloca a memória para a estrutura e o membro.

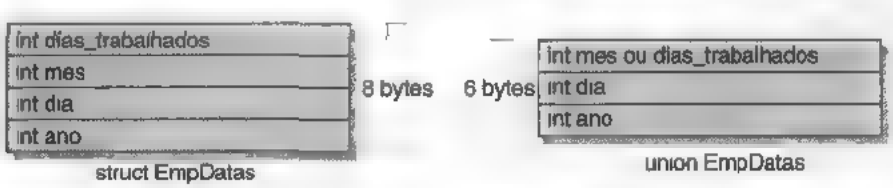

Figura 481 Alocando a memória para uma estrutura e uma união similares.

Como você aprenderá, o uso de uniões não somente economiza memória, mas também oferece aos seus programas a capacidade de interpretar os valores da memória de forma diferente.

ECONOMIZANDO MEMÓRIA COM AS UNIÕES 482

Na dica anterior você aprendeu que C lhe permite armazenar informações dentro de uma *união*. Quando você usa uma união, C aloca a quantidade de memória necessária para conter o maior membro da união. O programa a seguir, *uniaotam.c*, usa o operador *sizeof* para exibir a quantidade de memória que as diferentes uniões consomem:

```
#include <stdio.h>

void main(void)
 {
   union FuncionariosDatas
   {
     int dias_trabalhados;
     struct Data
     {
        int mes;
        int dia;
        int ano;
     } ultimo_dia;
   } emp_info;

   union Numeros
   {
      int a;
      float b;
      long c;
      double d;  // Maior -- requer 8 bytes
   } valor;

   printf("Tamanho de FuncionariosDatas %d bytes\n", sizeof(emp_info));
   printf("Tamanho dos Números %d bytes\n", sizeof(valor));
 }
```

Quando você compilar e executar o programa *uniaotam.c*, sua tela exibirá a seguinte saída:

```
Tamanho de FuncionariosDatas 6 bytes
Tamanho de Números 8 bytes
C:\>
```

USANDO REGS - UMA UNIÃO CLÁSSICA 483

Como você aprendeu, as uniões permitem que seus programas reduzam seus requisitos de memória e vejam as informações de formas diferentes. Na seção Serviços DOS e BIOS, mais à frente, você aprenderá que para acessar os serviços do DOS e da BIOS, seus programas normalmente atribuem parâmetros (no nível da linguagem assembly) a registradores específicos do microprocessador do PC. Para tornar os serviços do DOS e da BIOS disponíveis para seus programas, a maioria dos compiladores C oferece acesso às funções do DOS e da BIOS por meio das rotinas da biblioteca de execução, que usam uma união do tipo *REGS*:

```
struct WORDREGS
 {
   unsigned int ax, bx, cx, dx, si, di, cflag, flags;
 };
struct BYTEREGS
 {
   unsigned char al, ah, bl, bh, cl, ch, dl, dh;
 };
union REGS
 {
```

```
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

Quando seus programas acessam um dos registradores de propósito geral do microprocessador (AX, BX, CX e DX), o PC lhe permite referenciar os registradores em um formato de 16 bits. Alternativamente, você pode referenciar o byte menos significativo e o mais significativo (AL, AH, BL, BH, CL, CH, DL e DH). Como ambos os métodos referenciam o mesmo registrador, você tem dois modos de acessar a mesma posição de memória. Usando uma união, ambos os programas têm dois modos de acessar os registradores de propósito geral. A Figura 483 ilustra como C armazena as variáveis da união *REGS* na memória.

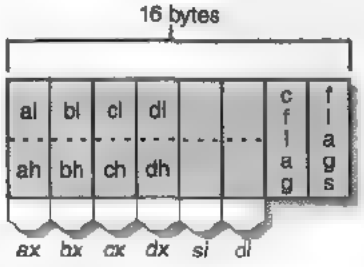

Figura 483 Como C armazena variáveis da união *REGS*.

484 COLOCANDO A UNIÃO REGS EM USO

Na Dica 483 você aprendeu que uma das uniões mais freqüentemente usadas nos programas baseados no DOS é a união *REGS*. O programa a seguir, *le_verx.c*, usa a união REGS para exibir a versão atual do DOS, acessando os registradores de propósito geral em sua forma de 16 bits:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   union REGS ent_regs, sai_regs;

   ent_regs.x.ax = 0x3000;
   intdos(&ent_regs, &sai_regs);
   printf("Versão atual %d.%d\n", sai_regs.x.ax & 0xFF,
       sai_regs.x.ax >> 8);
 }
```

O programa a seguir, *le_verh.c*, usa os registradores de 8 bits da união para exibir a versão atual do DOS:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   union REGS ent_regs, sai_regs;

   ent_regs.h.ah = 0x30;
   ent_regs.h.al = 0;
   intdos(&ent_regs, &sai_regs);
   printf("Versão atual %d.%d\n", sai_regs.h.al, sai_regs.h.ah);
 }
```

COMPREENDENDO AS ESTRUTURAS DE CAMPOS DE BIT 485

Muitas funções neste livro reduzem o número de variáveis (e, portanto, a quantidade de memória alocada) que seus programas precisam usar retornando valores cujos bits têm significados específicos. Quando os bits de um valor têm significados específicos, seus programas podem usar os operadores bit a bit de C para extrair os valores (os bits específicos). Por exemplo, assuma que seu programa precise controlar 100.000 datas. Você poderá criar uma estrutura do tipo *Data* para controlar as datas, como mostrado aqui:

```
struct Data
 {
   int mes;   // 1 até 12
   int dia;   // 1 até 31
   int ano;   // dois últimos dígitos
 };
```

Como uma alternativa, seus programas podem usar bits específicos dentro de um valor *unsigned int* para conter os campos de data, como mostrado na Figura 485.

Figura 485 Usando bits para representar uma data.

Depois disso, toda vez que seu programa precisar atribuir uma data, ele poderá efetuar as operações bit a bit corretas, como mostrado aqui:

```
unsigned data;

data = mes
data = data | (dia < 4);
data = data | (ano < 9);

printf("Mês %d Dia %d Ano %d\n", data & 0xF, (data > 4) & 0x1F, (data > 9));
```

No entanto, para tornar seus programas mais fáceis de compreender, C lhe permite criar uma *estrutura de campos de bits*. Ao declarar uma estrutura de campos de bits, você definirá uma estrutura que especifica o significado dos bits correspondentes:

```
struct Data
 {
   unsigned mes:4;
   unsigned dia:5;
   unsigned ano:7;
 } data;
```

Seus programas referenciarão então os campos de bit individualmente, como mostrado aqui:

```
data.dia = 31;
data.mes = 12;
data.ano = 94;

printf("Mês %d Dia %d Ano %d\n",  data.dia, data.mes, data.ano);
```

Nota: Quando você declara uma estrutura de campos de bits, todos os membros da estrutura precisam ser valores do tipo unsigned int.

VISUALIZANDO UMA ESTRUTURA DE CAMPOS DE BIT 486

Na dica anterior, você aprendeu que C lhe permite representar os bits dentro de um valor usando uma estrutura de campos de bit. Quando você declarar uma estrutura de campos de bit, C alocará bytes de memória suficientes para armazenar os bits da estrutura. Se a estrutura não usar todos os bits no último byte, a maioria dos compi-

ladores C inicializará os bits com 0. Para lhe ajudar a visualizar melhor como C armazena uma estrutura de campos de bit, a Figura 486 ilustra como o compilador C representará a estrutura de campos de bit, *Data*, como mostrado no código a seguir:

```
struct Data
 {
   unsigned mes:4;
   unsigned dia:5;
   unsigned ano:7;
 } data;
```

Figura 486 Como C representa a estrutura de campos de bits *Data*.

487 COMPREENDENDO O INTERVALO DE VALORES DE UMA ESTRUTURA BIT A BIT

Na Dica 486 vimos que C lhe permite representar bits dentro de um valor usando uma estrutura de campos de bit. Ao criar uma estrutura de campos de bit, você precisará alocar bits suficientes para armazenar o valor desejado de cada membro. Para lhe ajudar a determinar o número de bits requeridos, a Tabela 487 especifica o intervalo de valores que um dado número de bits pode representar.

Tabela 487 O intervalo de valores que seus programas podem representar com um determinado número de bits.

Tamanho do Campo	Intervalo de Valores
1	0-1
2	0-3
3	0-7
4	0-15
5	0-31
6	0-63
7	0-127
8	0-255
9	0-511
10	0-1023
11	0-2047
12	0-4095
13	0-8191
14	0-16383
15	0-32767
16	0-65535

488 PROCURANDO UM VALOR ESPECÍFICO EM UMA MATRIZ

Como você aprendeu, as matrizes lhe permitem armazenar valores do mesmo tipo. Algumas vezes, você pode querer procurar um valor específico em uma matriz. Existem dois modos comuns de pesquisar uma matriz: uma *pesquisa seqüencial* e uma *pesquisa binária*. Para efetuar uma pesquisa sequencial, seu programa inicia no primeiro elemento da matriz e procura um elemento de cada vez até encontrar o valor desejado ou até que o programa atinja o último elemento na matriz. Por exemplo, o laço *while* a seguir ilustra como seus programas podem procurar o valor 1.500 em uma matriz:

```
encontrado = 0;
i = 0;
while ((i  MATRIZ_ELEMENTOS) && (! encontrado))
```

```
   if (matriz[i] == 1500)
      encontrado = true;
   else
      i++;
if (i < MATRIZ_ELEMENTOS)
   printf("Valor encontrado no elemento %d\n", i);
else
   print("Valor não encontrado\n");
```

Se você anteriormente armazenou os valores em uma matriz de menor para o maior, seus programas podem efetuar uma pesquisa binária, que você aprenderá melhor na dica a seguir.

COMPREENDENDO UMA PESQUISA BINÁRIA 489

Como você aprendeu, um modo de localizar um valor dentro de uma matriz é procurar entre todos os elementos da matriz. Embora tal pesquisa seqüencial seja aceitável quando o tamanho da matriz é pequeno, percorrer em um laço uma matriz grande pode ser demorado. Caso seu programa já tenha classificado os valores na matriz do menor para o maior, ele pode usar uma *pesquisa binária* para localizar o valor. Esse tipo de pesquisa é chamado *pesquisa binária* porque, com cada operação, a pesquisa divide por dois o número de valores que precisa examinar.

O melhor modo de conceitualizar a pesquisa binária é pensar em como você consulta uma palavra em um dicionário. Assuma que você queira encontrar a palavra "Dálmata". Para começar, você pode abrir o dicionário no meio e examinar as palavras na página. Assumindo que você tenha aberto na letra M, sabe que "Dálmata" aparece antes da página atual, e, portanto, já elimina mais da metade das palavras no dicionário. Se você for para o meio das páginas restantes, muito provavelmente encontrará palavras que iniciam com a letra F. Novamente, você pode descartar metade das escolhas possíveis, e continuar sua pesquisa nas páginas que precedem a página atual. Desta vez, ao abrir no meio, provavelmente estará na letra C. A palavra "Dálmata" aparece em algum lugar nas páginas entre C e F. Quando você selecionar a página do meio, provavelmente estará nas palavras que iniciam com a letra D. Descartando páginas e selecionando a página do meio repetidamente, você pode chegar de forma rápida na página onde está a palavra "Dálmata"

Nota: Para efetuar uma pesquisa binária, seu programa precisa classificar os valores na matriz ou do menor para o maior ou do maior para o menor antes que você tente efetuar a pesquisa.

USANDO UMA PESQUISA BINÁRIA 490

Você já sabe que uma pesquisa binária fornece um modo rápido de procurar um determinado valor em uma matriz classificada. O programa a seguir, *binaria.c*, usa uma pesquisa binária para pesquisar vários valores na matriz *conta*, que contém os valores de 1 a 100. Para lhe ajudar a compreender melhor o processamento que a pesquisa binária realiza, a função *pesquisa_binaria* imprimirá mensagens que descrevem seu processamento:

```
#include <stdio.h>

int pesq_bin(int matriz[], int valor, int tamanho)
 {
   int achado = 0;
   int alto = tamanho, baixo = 0, med;

   med = (alto + baixo) / 2;
   printf("\n\nProcurando %d\n", valor);
   while ((! achado) && (alto >= baixo))
    {
      printf("Baixo %d Médio %d Alto %d\n", baixo, med, alto);
      if (valor == matriz[med])
        achado = 1;
      else if (valor < matriz[med])
        alto = med - 1;
      else
        baixo = med + 1;
      med = (alto + baixo) / 2;
```

```
    }
  return((achado) ? med: -1);
 }

void main(void)
 {
   int matriz[100], i;

   for (i = 0; i < 100; i++)
     matriz[i] = i;
   printf("Resultado da pesquisa %d\n", pesq_bin(matriz, 33, 100));
   printf("Resultado da pesquisa %d\n", pesq_bin(matriz, 75, 100));
   printf("Resultado da pesquisa %d\n", pesq_bin(matriz, 1, 100));
   printf("Resultado da pesquisa %d\n", pesq_bin(matriz, 1001, 100));
 }
```

Compile e execute o programa *binaria.c* e observe o número de operações que a pesquisa precisa executar para encontrar cada valor. O programa usa as variáveis *alto*, *meio* e *baixo* para controlar o intervalo de valores que está pesquisando atualmente.

491 CLASSIFICANDO UMA MATRIZ

Como você aprendeu, as matrizes lhe permitem armazenar valores relacionados do mesmo tipo. À medida que seus programas trabalharem com matrizes, algumas vezes você precisará classificar os valores de uma matriz, ou do menor para o maior (ordem ascendente) ou do maior para o menor (ordem descendente). Seus programas podem usar vários algoritmos de classificação diferentes para classificar as matrizes, incluindo o *método da Bolha*, da *seleção*, *Shell*, e *quick Sort*. Várias dicas a seguir discutem cada um desses métodos de classificação.

492 COMPREENDENDO O MÉTODO DA BOLHA

O *algoritmo (método) da bolha* é uma técnica simples de classificação de matrizes, sendo normalmente, o primeiro método que a maioria dos programadores aprende. Dada a sua simplicidade, o método da Bolha não é muito eficiente, e consome mais tempo do processador do que outras técnicas de classificação. No entanto, se você estiver classificando matrizes pequenas com 30 ou menos elementos, não haverá problema em usar o método da Bolha. Assumindo que você classifique valores do menor para o maior, o método da Bolha percorre os valores em uma matriz, comparando e movendo o maior valor para o final da matriz (como a bolha na água que sobe para a superfície). A Figura 492 ilustra quatro iterações do método da Bolha.

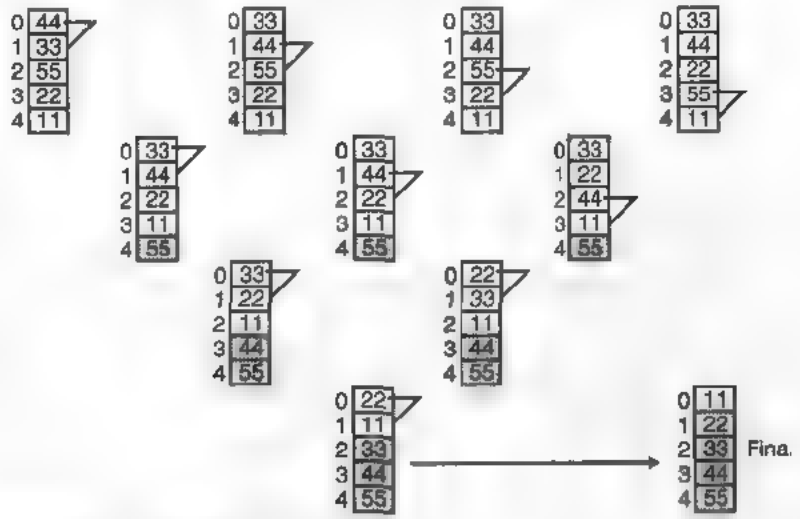

Figura 492 Quatro iterações do método da Bolha.

A primeira iteração move o valor mais alto para o final da matriz. A segunda iteração move o segundo maior valor para a penúltima posição na matriz. A terceira iteração move o terceiro maior valor, e assim por diante.

COLOCANDO O ALGORITMO DA BOLHA EM USO 493

A Dica 492 ilustrou rapidamente como o algoritmo da Bolha funciona. O programa a seguir, *bolha.c*, usa esse algoritmo para classificar uma matriz que contém 30 números aleatórios:

```
#include <stdio.h>
#include <stdlib.h>

void algoritmo_bolha(int matriz[], int tamanho)
 {
   int temp, i, j;

   for (i = 0; i < tamanho; i++)
    for (j = 0; j < tamanho; j++)
      if (matriz[i] < matriz[j])
        {
          temp = matriz[i];
          matriz[i] = matriz[j];
          matriz[j] = temp;
        }
 }

void main(void)
 {
   int valores[30], i;

   for (i = 0; i < 30; i++)
     valores[i] = rand() % 100;
   algoritmo_bolha(valores, 30);
   for (i = 0; i < 30; i++)
     printf("%d ", valores[i]);
 }
```

*Nota: A função **algoritmo_bolha** classifica valores do menor para o maior. Para inverter a ordem de classificação, simplesmente altere a comparação para if(matriz[i] > matriz[j]).*

COMPREENDENDO O ALGORITMO DA SELEÇÃO 494

O *algoritmo (método) da Seleção* é um método simples de classificação similar ao método da Bolha apresentado na dica anterior. Como o método da Bolha, seus programas somente devem usar o método de Seleção para classificar matrizes pequenas (30 elementos ou menos). O método da seleção se inicia selecionando um elemento da matriz (tal como o primeiro elemento). A classificação então pesquisa a matriz inteira até encontrar o valor mínimo. A classificação coloca o valor mínimo, seleciona o segundo elemento, e procura o segundo menor elemento. A Figura 494 ilustra duas iterações do método da Seleção em uma matriz de valores.

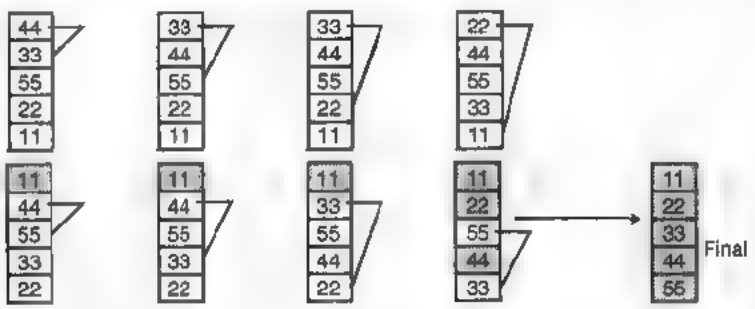

Figura 494 Classificando valores com o método da Seleção.

495 COLOCANDO EM USO O MÉTODO DA SELEÇÃO

A Dica 494 ilustrou rapidamente o funcionamento do algoritmo da Seleção. O programa a seguir, *selecao.c*, usa o algoritmo da Seleção para classificar uma matriz contendo 30 valores aleatórios.

```
#include <stdio.h>
#include <stdlib.h>

void algor_selecao(int matriz[], int tamanho)
 {
   int temp, atual, j;
   for (atual = 0; atual < tamanho; atual++)
    for (j = atual + 1; j < tamanho; j++)
      if (matriz[atual] > matriz[j])
        {
          temp = matriz[atual];
          matriz[atual] = matriz[j];
          matriz[j] = temp;
        }
 }

void main(void)
 {
   int valores[30], i;

   for (i = 0; i < 30; i++)
     valores[i] = rand() % 100;
   algor_selecao(valores, 30);
   for (i = 0; i < 30; i++)
     printf("%d ", valores[i]);
 }
```

Nota: A função algor_selecao classifica valores do menor para o maior. Para inverter a ordem de classificação, simplesmente altere a comparação para if(matriz[i] < matriz[j]).

496 COMPREENDENDO O ALGORITMO SHELL

O *algoritmo Shell* tem esse nome em homenagem ao seu criador, Donald Shell. A técnica de classificação Shell compara elementos da matriz separados por uma distância específica até que os elementos que ele compara com a distância atual estejam em ordem. O algoritmo Shell então divide a distância por dois e o processo continua. Quando a distância é finalmente 1, e nenhuma mudança ocorre, o algoritmo Shell completou seu processamento. A Figura 496 ilustra como o algoritmo Shell poderia classificar uma matriz.

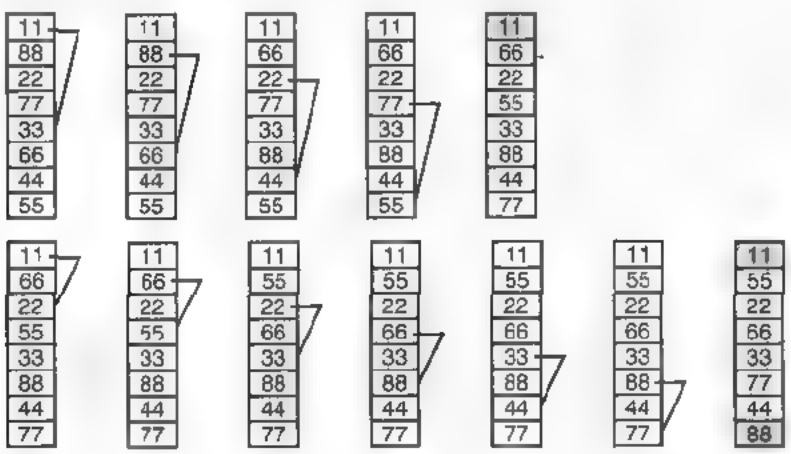

Figura 496 Classificando uma matriz como o algoritmo Shell.

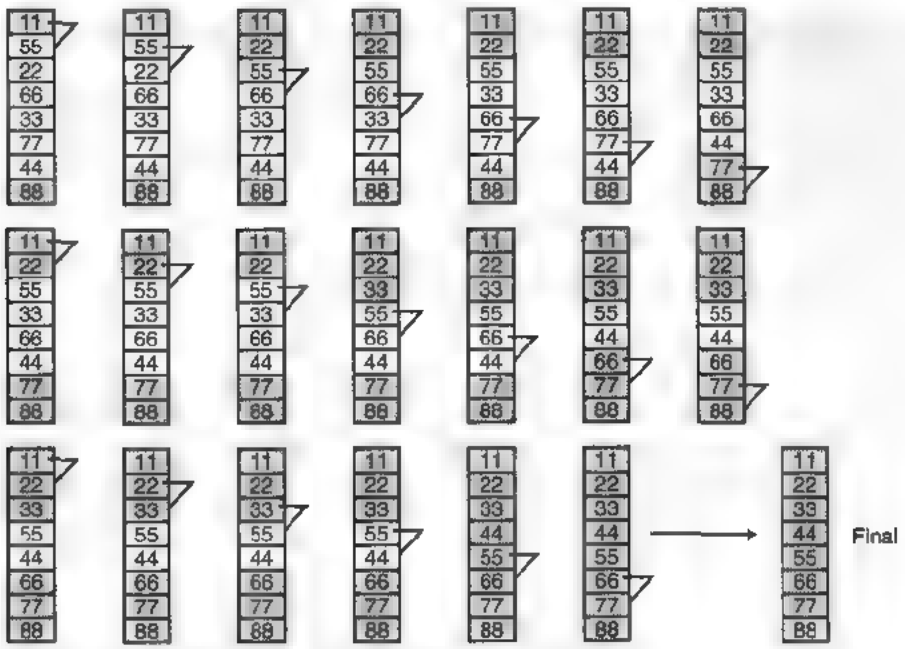

Figura 496 Classificando uma matriz como o algoritmo Shell (*Continuação*)

COLOCANDO EM USO O ALGORITMO SHELL 497

A dica anterior ilustrou rapidamente o funcionamento do algoritmo Shell. O programa a seguir, *shell.c*, usa o método Shell para classificar uma matriz contendo 50 valores aleatórios:

```
#include <stdio.h>
#include  <stdlib.h>

void algor_shell(int matriz[], int tamanho)
 {
   int temp, distancia, i, permuta_realizada;

   distancia = tamanho / 2;
   do
     {
       do
         {
           permuta_realizada = 0;
           for (i = 0; i < tamanho - distancia; i++)
           if (matriz[i] > matriz[i + distancia])
           {
             temp = matriz[i];
             matriz[i] = matriz[i + distancia];
             matriz[i + distancia] = temp;
             permuta_realizada = 1;
           }
         }
       while (permuta_realizada);
     }
   while (distancia = distancia / 2);
 }

void main(void)
 {
   int valores[50], i;
```

```
    for (i = 0; i < 50; i++)
      valores[i] = rand() % 100;
    algor_shell(valores, 50);
    for (i = 0; i < 50; i++)
      printf("%d ", valores[i]);
}
```

Nota: A função algor_selecao classifica valores do menor para o maior. Para inverter a ordem de classificação, simplesmente altere a comparação para if(matriz[i] < matriz[i + distancia]).

498 COMPREENDENDO O QUICK SORT

À medida que o número de elementos na sua matriz aumenta, o *Quick Sort* torna-se uma das técnicas mais rápidas que seus programas podem usar. O Quick Sort considera sua matriz como uma lista de valores. Quando a classificação inicia, ele seleciona o valor médio da lista como o *separador da lista*. A classificação então divide a lista em duas listas: uma com valores que são menores que o separador de lista, e uma segunda lista cujos valores são maiores ou iguais ao separador da lista. A classificação então chama recursivamente a si mesma com ambas as listas Toda vez que a classificação chama a si mesma, ela divide mais os elementos em listas menores. A Figura 498 ilustra como o Quick Sort poderia classificar uma matriz de 10 valores.

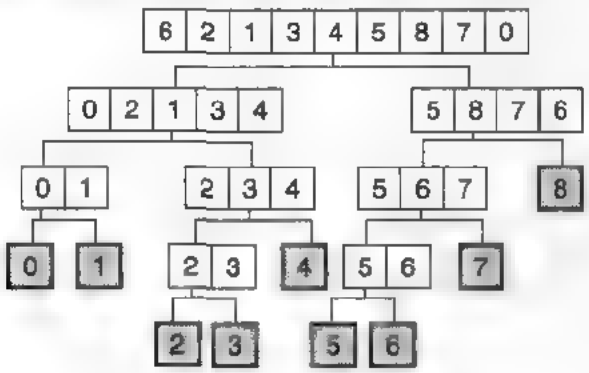

Figura 498 Classificando valores com o Quick Sort.

499 COLOCANDO O QUICK SORT EM USO

A Dica 498 ilustrou rapidamente como o Quick Sort funciona. O programa a seguir, *quick.c*, usa o algoritmo Quick Sort para classificar uma matriz que contém 100 números aleatórios:

```
#include <stdio.h>
#include <stdlib.h

void quick_sort(int matriz[], int primeiro, int ultimo)
 {
   int temp, baixo, alto, separador_lista;
   baixo = primeiro;
   alto = ultimo;
   separador_lista = matriz[(primeiro + ultimo) / 2];
   do
     {
       while (matriz[baixo] < separador_lista)
          baixo++;
       while (matriz[alto] > separador_lista)
          alto--;
       if (baixo <= alto)
         {
           temp = matriz[baixo];
           matriz[baixo++] = matriz[alto];
```

```
            matriz[alto--] = temp;
          }
      }
    while (baixo < = alto);
    if (primeiro < alto)
       quick_sort(matriz, primeiro, alto);
    if (baixo < ultimo)
      quick_sort(matriz, baixo, ultimo);
  }

void main(void)
  {
    int valores[100], i;

    for (i = 0; i < 100; i++)
      valores[i] = rand() % 100;
    quick_sort(valores, 0, 99);
    for (i = 0; i < 100; i++)
      printf("%d ", valores[i]);
  }
```

Nota: A função quick sort classifica valores do menor para o maior. Para inverter a ordem de classificação, simplesmente altere as comparações nos dois comandos while, como mostrado aqui:

```
while (matriz[baixo] > separador_lista)
    baixo++;

while (matriz[baixo] < separador_lista)
    alto++;
```

PROBLEMAS COM AS SOLUÇÕES DE CLASSIFICAÇÃO ANTERIORES 500

Várias dicas anteriores mostraram técnicas de classificação diferentes que seus programas podem usar para classificar as matrizes. No entanto, cada uma das dicas apresentadas trabalharam com matrizes do tipo *int*. Se seus programas precisarem classificar um tipo diferente de matriz, é necessário criar novas funções. Por exemplo, para classificar uma matriz do tipo *float*, seus programas precisam alterar o cabeçalho da função *quick_sort* e as declarações de variáveis, como mostrado aqui:

```
void quick_sort(float matriz[], int primeiro, int ultimo)
  {
    float temp, separador_lista;
    int baixo, alto;
```

Se você quiser classificar uma matriz de valores *long* mais tarde, precisará criar uma função diferente. No entanto, como você aprenderá, seus programas podem usar a função *qsort* da biblioteca de execução de C para classificar diferentes tipos de matriz. A função *qsort* usa indireção de memória para classificar valores de todos os tipos.

CLASSIFICANDO UMA MATRIZ DE STRINGS DE CARACTERES 501

Como você aprendeu, C lhe permite criar uma matriz de strings de caracteres, como mostrado aqui:

```
char *dias[] = { "Segunda", "Terça", "Quarta" };
```

Assim como algumas vezes seus programas precisam classificar matrizes de outros tipos, o mesmo é verdadeiro para classificar as matrizes de strings de caracteres. O programa a seguir, *str_clas.c*, usa um algoritmo da Bolha para classificar uma matriz de strings de caracteres:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

void bolha(char *matriz[], int tamanho)
 {
   char *temp;
   int i, j;

   for (i = 0; i < tamanho; i++)
    for (j = 0; j < tamanho; j++)
      if (strcmp(matriz[i], matriz[j]) < 0)
        {
          temp = matriz[i];
          matriz[i] = matriz[j];
          matriz[j] = temp;
        }
 }

void main(void)
 {
   char *valores[] = {"AAA", "CCC", "BBB", "EEE", "DDD"};
   int i;

   bolha(valores, 5);
   for (i = 0; i < 5; i++)
     printf("%s ", valores[i]);
 }
```

Quando a função classifica a matriz de strings de caracteres, a função não altera o conteúdo da string para reorganizar a matriz; em vez disso, ela organiza os ponteiros da string de caracteres para que as strings de caracteres estejam em ordem.

502 PESQUISANDO UMA MATRIZ COM LFIND

Como você aprendeu, uma operação de pesquisa seqüencial pesquisa os elementos de uma matriz em ordem até encontrar um valor específico. Para ajudar seus programas a pesquisar as matrizes de qualquer tipo, a biblioteca de execução de C fornece a função *lfind:*

```
#include <stdlib.h>
void *lfind(const void *elemento, void *base, size_t *num_itens, size_t
largura_elemento, int (*compare) (const void *, const void *));
```

Como você pode ver, a função utiliza muito os ponteiros. O parâmetro *elemento* é um ponteiro para o valor desejado. O parâmetro *base* é um ponteiro para o início da matriz. O parâmetro *num_itens* é um ponteiro para o número de elementos na matriz. O parâmetro *largura_elemento* especifica o número de bytes necessário para cada elemento da matriz. Finalmente, o parâmetro *compare* é um ponteiro para a segunda função que compara dois elementos da matriz. Ao contrário das funções mostradas anteriormente, que retornaram um índice de matriz para o valor desejado, a função *lfind* retorna um ponteiro para o valor desejado ou o valor 0, caso *lfind* não tenha encontrado o elemento. O programa a seguir, *lfind.c*, usa a função *lfind* para pesquisar um valor do tipo *int* e um valor do tipo *float:*

```
#include <stdlib.h>
#include <stdio.h>

int compare_int(int *a, int *b)
 {
   return(*a - *b);
 }

int compare_float(float *a, float *b)
```

```
{
  return((*a == *b) ? 0: 1);
}

void main(void)
{
  int int_valores[] = {1, 3, 2, 4, 5};
  float float_valores[] = {1.1, 3.3, 2.2, 4.4, 5.5};

  int *int_ptr, int_valor = 2, elementos = 5;
  float *float_ptr, float_valor = 33.3;
  int_ptr = lfind(&int_valor, int_valores, &elementos, sizeof(int),
    (int (*) (const void *, const void *)) compare_int);
  if (*int_ptr)
    printf("Valor %d encontrado\n", int_valor);
  else
    printf("Valor %d não encontrado\n", int_valor);
  float_ptr = lfind(&float_valor, float_valores, &elementos,
  sizeof(float),
    (int (*)(const void *,const void *)) compare_float);
  if (*float_ptr)
    printf("Valor %3.1f encontrado\n", float_valor);
  else
    printf("Valor %3.1f não-encontrado\n", float_valor);

}
```

Usando ponteiros, a função *lfind* pode eliminar os conflitos de tipos que afetaram as funções de pesquisa e de classificação discutidas anteriormente.

PROCURANDO VALORES COM LSEARCH 503

Na Dica 502 você aprendeu como usar a função *lfind* para pesquisar um elemento específico em uma matriz de valores. Se a função encontrasse o elemento, ela retornaria um ponteiro para ele. Se a função não encontrasse o elemento, ela retornaria 0 Dependendo dos seus programas, algumas vezes você desejará acrescentar o elemento na matriz se a função não o encontrar. Nesses casos, seus programas podem usar a função *lsearch*, como mostrado aqui:

```
#include <stdlib.h>

void *lsearch(const void *elemento, void *base, size_t *num_itens, size_t
largura_elemento, int (*compare) (const void *, const void *));
```

O programa a seguir, *lsearch.c*, usa a funçao *lsearch* para procurar o valor 1.500. Se *lsearch.c* não encontrar o valor, a função *lsearch* acrescentará o valor na matriz:

```
#include <stdlib.h>
#include <stdio.h

int compare_int(int *a, int *b)
{
  return(*a - *b);
}

void main(void)
{
  int int_valores[10] = {1, 3, 2, 4, 5};
  int *int_ptr, int_valor = 1500, elementos = 5, i;

  printf("Conteúdo da matriz antes da pesquisa\n");
  for (i = 0; i < elementos; i++)
```

```
    printf("%d ", int_valores[i]);
  int_ptr = lsearch(&int_valor, int_valores, &elementos, sizeof(int),
    (int (*) (const void *, const void *)) compare_int);
  printf("\nConteúdo da matriz após a pesquisa\n");
  for (i = 0; i < elementos; i++)
    printf("%d ", int_valores[i]);
}
```

Como você pode ver, quando a função acrescenta o valor na matriz, ela também atualiza o parâmetro de valor que especifica o número de elementos na matriz.

Nota: Quando seus programas usam a função lsearch, você precisa incluir espaço adicional dentro da matriz na qual pode acrescentar os valores

504 PESQUISANDO UMA MATRIZ CLASSIFICADA COM BSEARCH

Na Dica 489 foi visto que uma pesquisa binária localiza um valor em uma matriz classificada reduzindo repetidamente o número de elementos na matriz que ela pesquisará por um fator de 2 em cada iteração. Para ajudar seus programas a efetuar as operações de pesquisa binária, a biblioteca de execução de C oferece a função *bsearch:*

```
#include <stdlib.h>

void *bsearch(const void *chave, const void *base, size_t num_elementos,
size_t largura, int (*compare) (const void *, const void *));
```

Como a função *lsearch*, vista na dica anterior, a função *bsearch* utiliza muito os ponteiros. O parâmetro *elemento* é um ponteiro para o valor desejado. *base* é um ponteiro para o início da matriz. O parâmetro *num_itens* especifica o número de elementos na matriz. O parâmetro *largura_elemento* especifica o número de bytes necessários para cada elemento da matriz. Finalmente, o parâmetro *compare* é um ponteiro para uma segunda função, que compara dois elementos da matriz. Ao contrário das funções mostradas anteriormente, que retornaram o índice de matriz do valor dese ado, a função *bsearch* retorna um ponteiro para o valor desejado, ou o valor 0 se a função *bsearch* não encontra o elemento. O programa a seguir, *bsearch.c*, usa a função *bsearch* para pesquisar duas matrizes diferentes, uma para um valor do tipo *int* e uma para um valor do tipo *float:*

```
#include <stdlib.h>
#include <stdio.h>

int compare_int(int *a, int *b)
 {
   return(*a - *b);
 }

int compare_float(float *a, float *b)
 {
   return((*a == *b) ? 0: 1);
 }

void main(void)
 {
   int int_valores[] = {1, 3, 2, 4, 5};
   float float_valores[] = {1.1, 3.3, 2.2, 4.4, 5.5};
   int *int_ptr, int_valor = 2, elementos = 5;
   float *float_ptr, float_valor = 33.3;

   int_ptr = bsearch(&int_valor, int_valores, elementos, sizeof(int),
     (int (*) (const void *,const void *)) compare_int);
   if (*int_ptr)
     printf("Valor %d encontrado\n", int_valor);
   else
      printf("Valor %d não-encontrado\n", int_valor);
   float_ptr = bsearch(&float_valor, float_valores, elementos, sizeof(float),
     (int(*) (const void *,const void *)) compare_float);
```

```
  if (*float_ptr)
     printf("Valor %3.1f encontrado\n", float_valor);
  else
     printf("Valor %3.1f não encontrado\n", float_valor);

}
```

Nota: Para usar a função bsearch, os valores da matriz precisam estar classificados do menor para o maior.

CLASSIFICANDO MATRIZES COM QSORT 505

Na Dica 498 você aprendeu que o algoritmo quick sort classifica os elementos da matriz tratando a matriz como uma lista. Como o Quick Sort repetidamente separa os elementos em listas classificadas menores, ele é muito eficiente. Para ajudar seus programas a classificar as matrizes de todos os tipos usando um Quick Sort, a biblioteca de execução de C oferece a função *qsort*, como mostrado aqui:

```
#include <stdlib.h>

void *qsort(void *base, size_t num_itens, size_t larg_elem,
     int (*compare)(const void *, const void *));
```

Como as funções *lsearch* e *bsearch* que você já aprendeu, a função *qsort* utiliza muito os ponteiros. O parâmetro *base* é um ponteiro para o início da matriz. *num_itens* especifica o número de elementos na matriz. O parâmetro *larg_elem* especifica o número de bytes necessários para cada elemento da matriz. Finalmente, o parâmetro *compare* é um ponteiro para uma segunda função, que compara dois elementos de matriz e retorna um valor, como mostrado aqui:

```
*a < *b     // Retorna valor < 0
*a == *b    // Retorna 0
*a > *b     // Valor > 0
```

O programa a seguir, *qsort.c*, usa a função *qsort* para procurar um valor do tipo *int* e um valor do tipo *float*:

```
#include <stdlib.h>
#include <stdio.h>

int compare_int(int *a, int *b)
 {
   if (*a < *b)
      return(-1);
   else if (*a == *b)
      return(0);
   else
      return(1);
 }

int compare_float(float *a, float *b)
 {
   if (*a < *b)
     return(-1);
   else if (*a == *b)
     return(0);
   else
     return(1);
 }

void main(void)
 {
   int int_valores[] = {51, 23, 2, 44, 45};
   float float_valores[]={21.1, 13.3, 22.2, 34.4, 15.5};
   int elementos = 5, i;
```

```
  qsort(int_valores, elementos, sizeof(int),
    (int (*) (const void *, const void *)) compare_int);
  for (i = 0; i < elementos; i++)
    printf("%d ", int_valores[i]);
  putchar('\n');
  qsort(float_valores, elementos, sizeof(float),
    (int(*) (const void *,const void *)) compare_float);
  for (i = 0; i < elementos; i++)
    printf("%4.1f ", float_valores[i]);
}
```

506 DETERMINANDO O NÚMERO DE ELEMENTOS NA MATRIZ

Várias dicas anteriores incluíram o número de elementos da matriz como um parâmetro da função. Se o número de elementos na sua matriz puder mudar, você poderá reduzir o número de alterações que precisa fazer no seu programa usando um valor constante, tal como o seguinte:

```
#define NUM_ELEMENTOS = 5
```

Alternativamente, seus programas podem usar o operador *sizeof* para determinar o número de elementos em uma matriz, como mostrado aqui:

```
elementos = sizeof(matriz) / sizeof(matriz[0]);
```

O programa a seguir, *num_els.c*, usa o operador *sizeof* para derivar o número de elementos nas duas matrizes, e, depois, exibe o número de elementos nessas matrizes:

```
#include <stdio.h>

void main(void)
{
  int int_valores[] = {51, 23, 2, 44, 45};
  float float_valores[] = {21.1, 13.3, 22.2, 34.4, 15.5};

  printf("Número de elementos em int valores %d\n",
     sizeof(int_valores) / sizeof(int_valores[0]));
  printf("Número de elementos em float_valores %d\n",
     sizeof(float_valores) / sizeof(float_valores[0]));
}
```

507 COMPREENDENDO PONTEIROS COMO ENDEREÇOS

Como visto na seção Introdução à Linguagem C, uma variável é o nome de uma posiçao na memória que pode armazenar um valor de um determinado tipo. Seu programa referencia cada posição na memória usando um *endereço* exclusivo. Um *ponteiro* é uma variável ou um valor que contém um endereço. A linguagem de programação C utiliza muito os ponteiros. Quando você passa matrizes ou strings para as funções, o compilador C passa um ponteiro. Da mesma forma, quando uma função precisa alterar o valor de um parâmetro, o programa deve passar para a função um ponteiro para o endereço de memória da variável. Várias dicas a seguir examinam os ponteiros em detalhe.

508 DETERMINANDO O ENDEREÇO DE UMA VARIÁVEL

Um *ponteiro* é um endereço de uma posição na memória. Quando seus programas trabalham com matrizes (e strings), o programa trabalha com um ponteiro para o primeiro elemento da matriz. Quando seus programas precisarem determinar o endereço de uma variável, deverão usar o operador de *endereço* de C (&). Por exemplo, o programa a seguir, *endereco.c*, usa o operador de *endereço* para exibir o endereço de várias variáveis diferentes:

```
#include <stdio.h>

void main(void)
```

```
{
  int conta = 1;
  float salario = 40000.0;
  long distancia = 1234567L;

  printf("O endereço de conta é %x\n", &conta);
  printf("O endereço de salario é %x\n", &salario);
  printf("O endereço de distancia é %x\n", &distancia);
}
```

Quando você compilar e executar o programa *endereco.c*, sua tela exibirá a seguinte saída:

```
O endereço de conta é fff4
O endereço de salario é fff0
O endereço de distancia é ffec
C:\
```

COMPREENDENDO COMO C TRATA MATRIZES COMO PONTEIROS 509

Já vimos que, o compilador C trata as matrizes como ponteiros. Por exemplo, quando seu programa passa uma matriz para uma função, o compilador passa o endereço inicial da matriz. O programa a seguir, *somamat.c*, exibe o endereço inicial de várias matrizes diferentes:

```
#include <stdio.h>

void main(void)
{
  int conta[10];
  float salarios[5];
  long distancia[10];

  printf("O endereço da matriz conta é %x\n", conta);
  printf("O endereço da matriz salarios é %x\n", salarios);
  printf("O endereço da matriz distancias é %x\n", distancias);
}
```

Quando você compilar e executar o programa *somamat.c*, sua tela exibirá a seguinte saída:

```
O endereço da matriz conta é ffe2
O endereço da matriz salarios é ffce
O endereço da matriz distancias é ffa6
C:\>
```

APLICANDO O OPERADOR DE ENDEREÇO (&) A UMA MATRIZ 510

Como você aprendeu, o compilador C trata uma matriz como um ponteiro para o primeiro elemento da matriz. A Dica 508 mostrou que C usa o operador de endereço (&) para retornar o endereço de uma variável. Se você aplicar o operador de endereço a uma matriz, C retornará o endereço inicial da matriz. Portanto, aplicar o operador de endereço a uma matriz é redundante. O programa a seguir, *matriz2.c*, exibe o endereço inicial de uma matriz, seguido pelo ponteiro que o operador de endereço de C retorna:

```
#include <stdio.h>

void main(void)
{
  int conta[10];
  float salarios[5];
  long distancias[10];
```

```
    printf("O endereço da matriz conta é %x &conta é %x\n", conta, &conta);
    printf("O endereço da matriz salarios é %x &salários é %x\n",
       salarios, &salarios);
    printf("O endereço da matriz distancias é %x &distancias é %x\n",
       distancias, &distancias);
 }
```

Quando você compilar e executar o programa *matriz2.c*, sua tela exibirá o seguinte resultado:

```
O endereço da matriz conta é ffe2 &conta é ffe2
O endereço da matriz salarios é ffce &salarios é ffce
O endereço da matriz distancias é ffa6 &distancias é ffa6
C:\>
```

511 DECLARANDO VARIÁVEIS PONTEIRO

À medida que seus programas se tornarem mais complexos, você usará os ponteiros com muita frequência. Para armazenar ponteiros, seus programas precisam declarar variáveis ponteiro. Para declarar um ponteiro, você precisa especificar o tipo do valor ao qual o ponteiro aponta (tal como *int*, *float*, *char* etc.) e um asterisco (*) antes do nome da variável. Por exemplo, o comando a seguir declara um ponteiro para um valor do tipo *int*.

```
int *iptr;
```

Como qualquer variável, você precisa atribuir um valor a uma variável ponteiro antes de poder usar o ponteiro dentro de seu programa. Quando você atribui um valor a um ponteiro, realmente atribui um endereço. Assumindo que você tenha anteriormente declarado *int conta*, o comando a seguir atribui o endereço da variável *conta* ao ponteiro *iptr*:

```
iptr = &conta;    // Atribui o endereço de conta a iptr
```

O programa a seguir, *iptr.c*, declara a variável-ponteiro *iptr* e atribui ao ponteiro o endereço da variável *conta*. O programa então exibe o valor da variável ponteiro, juntamente com o endereço de *conta*:

```
#include <stdio.h>

void main(void)
 {
   int *iptr;       // Declara variável ponteiro
   int conta = 1;

   iptr = &conta;
   printf("Valor de iptr %x Valor de conta %d Endereço de conta %x\n",
     iptr, conta, &conta);
 }
```

Quando você compilar e executar o programa *iptr.c*, sua tela exibirá o seguinte resultado:

```
Valor de iptr fff2 Valor de conta 1 Endereço de conta fff2
C:\>
```

512 DESREFERENCIANDO UM PONTEIRO

Como você aprendeu, um ponteiro contém um endereço que aponta para um valor de um tipo específico. Usando o endereço que um ponteiro contém, você pode determinar o valor na memória para o qual o ponteiro aponta. *Desreferenciar um ponteiro* é o processo de acessar o valor em uma posição de memória específica. Para desreferenciar o valor de um ponteiro, você usa o operador *asterisco de indireção* (*). Por exemplo, o comando *printf* a seguir exibe o valor apontado pelo ponteiro inteiro *iptr*:

```
printf("O valor apontado por iptr é %d\n", *iptr);
```

Da mesma forma, o comando a seguir atribui o valor apontado pela variável *iptr* para a variável *conta*:

```
conta = *iptr;
```

Finalmente, o comando a seguir atribui o valor 7 à posição de memória apontada por *iptr*:

```
*iptr = 7;
```

Nota: Para usar o valor armazenado na posição de memória apontada por um ponteiro, você precisa desreferenciar o valor do ponteiro usando o operador asterisco de indireção (*).

USANDO VALORES DE PONTEIRO 513

A Dica 510 mostrou que é possíve. atribuir um endereço a uma variável ponteiro usando o operador de *endereço* (&). Na Dica 512 vimos que, para acessar o valor armazenado na memória na posição de memória apontada por um ponteiro, você precisa usar o operador de *indireção asterisco* (*). O programa a seguir, *ptr_demo.c*, atribui ao ponteiro *int iptr* o endereço da variável *conta*. O programa depois exibe o valor do ponteiro e o valor armazenado na posição apontada pelo ponteiro (o valor *contador*). O programa então modifica o valor apontado pelo ponteiro, como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   int contador = 10;
   int *iptr;           // Declara valor do ponteiro

   iptr = &contador;    // Atribui o endereço
   printf("Endereço em iptr %x Valor em *iptr %d\n", iptr, *iptr);
   *iptr = 25;          // Altera o valor na memória
   printf("Valor de contador %d\n", contador);
 }
```

USANDO PONTEIROS COM PARÂMETROS DE FUNÇÃO 514

A seção Funções deste livro examina em detalhe o processo de passar parâmetros para as funções. Como você aprenderá, quando quiser alterar o valor de um parâmetro, deverá passar para a função um ponteiro para um parâmetro. O programa a seguir, *trocaval.c*, usa os ponteiros para dois parâmetros do tipo *int* para permutar os valores das variáveis, como mostrado aqui:

```
#include <stdio.h>

void troca_valores(int *a, int *b)
 {
   int temp;

   temp = *a;  // Armazena temporariamente o valor apontado por a
   *a = *b;    // Atribui o valor de b a a
   *b = temp;  // Atribui o valor de a a b
 }

void main(void)
 {
   int um = 1, dois = 2;

   troca_valores(&um, &dois);
   printf("um contém %d dois contém %d\n", um, dois);
 }
```

Como você pode ver, dentro da função, os comandos desreferenciam os ponteiros usando o operador de *indireção* (*). O programa passa o endereço de cada variável para a função usando o operador de *endereço* (&).

515 COMPREENDENDO A ARITMÉTICA DE PONTEIROS

Um ponteiro é um endereço que aponta para um valor de um determinado tipo na memória. Nos termos mais simples possíveis, um ponteiro é um valor que aponta para uma posição de memória específica. Se você somar o valor 1 a um ponteiro, o ponteiro apontará para a próxima posição na memória. Se você somar 5 ao valor de um ponteiro, o ponteiro apontará para a posição de memória cinco posições adiante do endereço atual. No entanto, a aritmética de ponteiro não é tão simples quanto você pode imaginar. Por exemplo, assuma que um ponteiro contenha o endereço 1000. Se você somasse 1 ao ponteiro, poderia esperar que o resultado fosse 1001. No entanto, o endereço resultante depende do tipo do ponteiro. Por exemplo, se você somar 1 a uma variável ponteiro do tipo *char* (que contém 1000), o endereço resultante será 1001. Se você somar 1 a um ponteiro do tipo *int* (que requer dois bytes na memória), o endereço resultante será 1002. Quando você efetuar aritmética de ponteiro, tenha em mente o tipo do ponteiro. Além de somar valores aos ponteiros, seus programas poderão subtrair valores ou somar e subtrair dois ponteiros. Muitas dicas nesta seção apresentam várias operações de aritmética de ponteiros.

516 INCREMENTANDO E DECREMENTANDO UM PONTEIRO

À medida que seus programas trabalharem com ponteiros, uma das operações mais comuns que eles efetuarão será incrementar e decrementar o valor de um ponteiro para apontar para a próxima posição ou para posição anterior na memória. O programa a seguir, *pontmatz.c*, atribui o endereço inicial de uma matriz de valores inteiros ao ponteiro *iptr*. O programa depois incrementa o valor do ponteiro para exibir os cinco elementos que a matriz contém:

```
#include <stdio.h>

void main(void)
 {
   int valores[5] = {1, 2, 3, 4, 5};
   int contador;
   int *iptr;

   iptr = valores;
   for (contador = 0; contador < 5; contador++)
     {
       printf("%d\n", *iptr);
       iptr++;
     }
 }
```

Quando você compilar e executar o programa *pontmatz.c*, sua tela exibirá os valores de 1 a 5. O programa inicialmente atribui o endereço inicial da matriz ao ponteiro. Em seguida, o programa incrementa o ponteiro para apontar para cada elemento.

517 COMBINANDO UMA REFERÊNCIA E INCREMENTO DE PONTEIRO

Na Dica 516 você usou o ponteiro *iptr* para exibir o conteúdo de uma matriz. Para exibir o conteúdo da matriz, o ponteiro usou um laço *for*, como mostrado aqui:

```
for (contador = 0; contador < 5; contador++)
 {
   printf("%d\n", *iptr);
   iptr++;
 }
```

Como você pode ver, o laço *for* acessa o valor do ponteiro em uma linha, e, depois, incrementa o ponteiro na próxima linha. Você já aprendeu que pode usar o operador de *incremento de sufixo* de C para usar o valor de uma variável, e, depois, incrementar o valor.

O laço *for* a seguir usa o operador de incremento de sufixo para referenciar o valor apontado pela variável ponteiro e depois incrementa o valor do ponteiro:

```
for (contador = 0; contador < 5; contador++)
   printf("%d\n", *iptr++);
```

PERCORRENDO UMA STRING USANDO UM PONTEIRO 518

A seção Compreendendo as Strings deste livro utiliza muito os ponteiros. Como você aprendeu, uma string é uma matriz de caracteres terminada por *NULL*. O programa a seguir, *str_ptr.c*, usa a função *exibe_string* para exibir uma *string* de caracteres usando um ponteiro:

```
#include <stdio.h>

void exibe_string(char *string)
 {
   while (*string)
     putchar(*string++);
 }

void main(void)
 {
   exibe_string("Bíblia do Programador C/C++");
 }
```

Como você pode ver, a função *exibe_string* declara a variável string como um ponteiro. Usando o ponteiro, a função simplesmente percorre os caracteres da string até encontrar o caractere *NULL*. Para exibir o caractere, a função *exibe_string* primeiro desreferencia o endereço do ponteiro (obtendo o caractere). Em seguida, a função incrementa o ponteiro para apontar para o próximo caractere na string.

USANDO FUNÇÕES QUE RETORNAM PONTEIROS 519

Vimos que as funções podem retornar um valor para seus programas. O valor que uma função retorna é sempre do tipo declarado no protótipo ou cabeçalho da função (tal como *int*, *float* ou *char*). Além de retornar esses tipos básicos, as funções podem declarar ponteiros para os valores. Por exemplo, a função *fopen*, que a maioria dos programas C usa para abrir um canal de arquivo, retorna um ponteiro para uma estrutura do tipo *FILE*, como mostrado aqui:

```
FILE *fopen(const char *nomecaminho, const char *modo);
```

De um modo similar, muitas funções apresentadas na seção Compreendendo as Strings retornam ponteiros para as strings de caracteres. À medida que você examinar os protótipos de funções apresentadas, observe as funções que retornam um *ponteiro para um valor*, em vez de um valor de um tipo básico.

CRIANDO UMA FUNÇÃO QUE RETORNA UM PONTEIRO 520

Na dica anterior você aprendeu que muitas funções da biblioteca de execução de C retornam ponteiros. À medida que seus programas se tornarem mais complexos, você criará funções que retornam ponteiros para tipos específicos. Por exemplo, o programa a seguir, *ptr_mais.c*, cria uma função chamada *string_maiusc* que converte todos os caracteres de uma string para maiúsculas e depois retorna um ponteiro para a string:

```
#include <stdio.h>
#include <ctype.h>

char *string_maiusc(char *string)
 {
   char *ender_inicial, *temp;
   ender_inicial = temp = string;
   while (*string)
      *(temp++) = toupper(*string++);
   return(ender_inicial);
 }
```

```
void main(void)
 {
   char *titulo = "Bíblia do Programador C/C++!";
   char *string;

   string = string_maiusc(titulo);
   printf("%s\n", string);
   printf("%s\n", string_maiusc("Matrizes e Ponteiros"));
 }
```

Como você pode ver, para criar uma função que retorne um ponteiro, simplesmente coloque o asterisco antes do nome da função, como mostrado aqui:

```
char *string_maiusc(char *string);
```

521 COMPREENDENDO UMA MATRIZ DE PONTEIROS

Várias dicas apresentadas nesta seção discutiram as matrizes detalhadamente. Até aqui, todas as matrizes usaram os tipos básicos de C (tais como *int*, *float* e *char*); no entanto, C não restringe as matrizes a esses tipos simples. Assim como você pode criar funções que retornam ponteiros, também pode criar matrizes de ponteiros. Você mais comumente usará matrizes para conter strings de caracteres. Como um exemplo, a declaração a seguir cria uma matriz, chamada *dias*, que contém ponteiros para strings de caracteres:

```
char *dias[7]= { "Domingo", "Segunda", "Terça", "Quarta",
                 "Quinta", "Sexta", "Sábado" };
```

Se você examinar o tipo da matriz da direita para a esquerda, verá que a matriz contém sete elementos. O asterisco antes do nome da variável especifica um ponteiro. Se você combinar o nome de tipo *char* que precede o nome da variável, a declaração se tornará uma matriz de ponteiros para strings de caractere (no exemplo anterior existem sete strings). Uma das matrizes de ponteiros para strings de caracteres mais amplamente usadas é *argv*, que contém a linha de comando de seu programa, como detalhado na seção Redirecionando a E/S e Processando Linhas de Comando, mais à frente.

Nota: Quando você declara uma matriz de ponteiros para strings de caracteres, o compilador C não inclui um item NULL para indicar o final da matriz, como faz para as strings de caracteres.

522 VISUALIZANDO UMA MATRIZ DE STRINGS DE CARACTERES

Como você aprendeu, C trata uma matriz como um ponteiro para a posição inicial da matriz na memória. Na dica anterior você criou uma string de caracteres chamada *dias* que continha os dias da semana. Quando você cria uma matriz de strings de caracteres, o compilador C armazena ponteiros para as strings da matriz. A Figura 522 ilustra como o compilador C armazenaria a matriz *letras*, como mostrado aqui:

```
char *letras[4] = {"AAA", "BBB", "CCC", "DDD"};
```

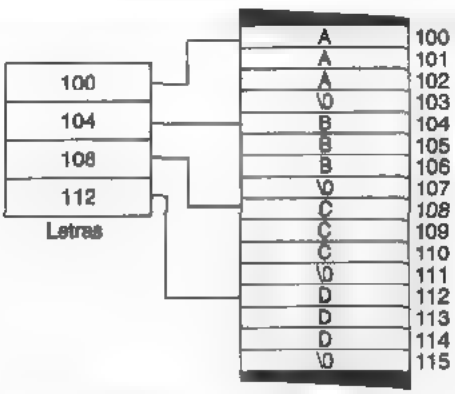

Figura 522 C armazena matrizes de strings de caracteres como uma matriz de ponteiros.

Nota: Quando você declara uma matriz de strings de caracteres, o compilador C não acrescenta um item NULL para indicar o final da matriz como faz para as strings de caracteres.

PERCORRENDO EM UM LAÇO UMA MATRIZ DE STRINGS DE CARACTERES 523

Ao criar uma matriz de strings de caracteres, C armazena os ponteiros para cada string dentro dos elementos da matriz. O programa a seguir, *dias.c*, percorre em um laço a matriz *dias*, que contém ponteiros para strings que contêm os nomes dos dias da semana, como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   char *dias[7]={"Domingo", "Segunda", "Terça",
                  "Quarta", "Quinta", "Sexta", "Sábado");
   int i;

   for (i = 0; i < 7; i++)
     printf("dias[%d] contém %s\n", i, dias[i]);
 }
```

Como você pode ver, o programa simplesmente percorre em um laço os elementos da matriz, usando o especificador de formato %*s* de *printf*.

TRATANDO UMA MATRIZ DE STRING DE CARACTERES COMO UM PONTEIRO 524

Como você aprendeu, C trata uma matriz como um ponteiro para o elemento inicial da matriz na memória. Várias dicas apresentadas na seção Strings deste livro acessam as matrizes de string de caracteres usando um ponteiro que é similar ao seguinte:

```
char *string;
```

Você também aprendeu que C lhe permite criar matrizes de strings de caracteres. Por exemplo, a declaração a seguir cria uma matriz chamada *diasuteis*, que pode armazenar os ponteiros para cinco strings de caracteres:

```
char *diasuteis[5];
```

Como a declaração cria uma matriz, C lhe permite acessar a matriz usando um ponteiro. Para acessar a matriz usando um ponteiro, você precisa declarar uma variável ponteiro que aponte para uma matriz de strings de caracteres. No caso da matriz *diasuteis*, a declaração do ponteiro de referência seria a seguinte:

```
char **dia_util_ptr;
```

Os asteriscos duplos, neste caso, especificam que *dia_util_ptr* é um *ponteiro para um ponteiro para uma string de caracteres*. Várias dicas apresentadas na seção Redirecionando a E/S e Processando Linhas de Comando mais à frente, trabalham com um ponteiro para um ponteiro para strings de caracteres.

USANDO UM PONTEIRO PARA UM PONTEIRO PARA STRINGS DE CARACTERES 525

Na Dica 524 você aprendeu que a declaração a seguir cria um ponteiro para uma string de caracteres:

```
char **dia_util_ptr;
```

O programa C a seguir, *diautil.c*, usa um ponteiro para um ponteiro para strings de caracteres para exibir o conteúdo da matriz *diasuteis*:

```
#include <stdio.h>

void main(void)
 {
   char *diasuteis[] = {"Segunda", "Terça", "Quarta", "Quinta", "Sexta", "" };
   char **dia_util;

   dia_util = diasuteis;
   while (*dia_util)
     printf("%s\n", *dia_util++);
 }
```

Quando o programa inicia, ele atribui ao ponteiro *dia_util* o endereço inicial da matriz *diasuteis* (o endereço da string Segunda). O programa então repete um laço até encontrar o ponteiro para a string *NULL* (a condição final).

Nota: Quando você declara uma matriz de strings de caracteres, o compilador C não acrescenta um caractere NULL para indicar o final da matriz — como faz para as strings de caracteres. Portanto, a declaração da matriz diasuteis explicitamente incluiu uma string NULL para que o programa possa testar dentro do laço.

526 DECLARANDO UMA CONSTANTE STRING USANDO UM PONTEIRO

Várias dicas apresentadas até aqui inicializaram strings de caracteres na declaração, como mostrado aqui:

```
char titulo[] = "Bíblia do Programador C/C++";
```

Quando você declara uma matriz com colchetes vazios, o compilador C aloca memória suficiente para armazenar os caracteres especificados (e o terminador *NULL*), atribuindo à variável *titulo* um ponteiro para o primeiro caractere. Como o compilador C automaticamente aloca a memória necessária e depois trabalha com um ponteiro para a memória, seus programas podem usar um ponteiro de string de caracteres, em vez de uma matriz, como mostrado aqui:

```
char *titulo = "Bíblia do Programador C/C++";
```

527 COMPREENDENDO O PONTEIRO DO TIPO VOID

Como você aprendeu, ao declarar uma variável ponteiro, é preciso especificar o tipo do valor para o qual o ponteiro aponta (tal como *int*, *float*, ou *char*). Quando você faz isso, o compilador pode, mais tarde, efetuar aritmética de ponteiros corretamente e adicionar os valores de deslocamento corretos quando você incrementar ou decrementar o ponteiro. Em alguns casos, no entanto, seus programas não manipularão o valor de um ponteiro de qualquer maneira. Em vez disso, seus programas somente irão querer obter um ponteiro para uma posição de memória com a qual o programa determinará o uso do ponteiro. Nesses casos, seus programas poderão criar um ponteiro para o tipo *void*, como mostrado aqui:

```
void *ponteiro_memória;
```

À medida que você for examinando as funções da biblioteca de execução de C apresentadas na seção Gerenciamento Memória no Windows, mais à frente, deste livro, verá que várias funções retornam ponteiros para o tipo *void*. Essas funções basicamente informam que retornam um ponteiro para uma posição de memória sobre a qual o compilador não faz suposições quanto ao conteúdo ou modo de acesso da memória.

528 CRIANDO PONTEIROS PARA AS FUNÇÕES

Você sabe que C lhe permite criar ponteiros para todos os tipos de dados (tais como *int*, *char*, *float* e até strings de caracteres). Além disso, C permite que seus programas criem e usem ponteiros para as funções. O uso mais comum dos ponteiros para funções é permitir que seus programas passem uma função como um parâmetro para outra função. As seguintes declarações criam ponteiros para funções:

```
int (*min)();
int (*max)();
float (*media)();
```

Observe o uso dos parênteses entre os nomes das variáveis. Se você removesse os parênteses, as declarações serviriam como protótipos de função para as funções que retornam ponteiros para um tipo específico, como mostrado aqui:

```
int *min();
int *max();
float *media;
```

Ao ler uma declaração de variável, comece com a declaração mais interna que aparece dentro dos parênteses, e, depois, trabalhe da direita para a esquerda:

```
int (*min)();
```

USANDO UM PONTEIRO PARA UMA FUNÇÃO 529

Na Dica 528 você aprendeu que C lhe permite criar ponteiros para as funções. O uso mais comum de um ponteiro para uma função é passar essa função como um parâmetro para outra função. Anteriormente, nesta seção, você examinou as funções de classificação e de pesquisa da biblioteca de execução de C. Como você sabe, se quiser classificar valores do menor para o maior, você passará uma função específica para a rotina da biblioteca de execução. Se você quiser classificar valores do maior para o menor, passará uma função diferente. O programa a seguir, *passfunc.c*, passa a função *min* ou *max* para a função *pega_result*. Dependendo da função que o programa passa, o valor que *pega_result* retorna será diferente:

```
#include <stdio.h>

int pega_result(int a, int b, int (*compare)())
 {
   return(compare(a, b));  // Chama a função passada
 }

int max(int a, int b)
 {
   printf("Em max\n");
   return((a > b) ? a: b);
 }

int min(int a, int b)
 {
   printf("Em min\n");
   return((a < b) ? a: b);
 }

void main(void)
 {
   int result;

   result = pega_result(1, 2, &max);
   printf("O máximo entre 1 e 2 é %d\n", result);
   result = pega_result(1, 2, &min);
   printf("O mínimo de 1 e 2 é %d\n", result);
 }
```

530 USANDO UM PONTEIRO PARA UM PONTEIRO PARA UM PONTEIRO

C lhe permite criar variáveis que são ponteiros para outros ponteiros. Em geral, não há limite no número de *indireções* (ponteiros para ponteiros) que seus programas podem usar. No entanto, para a maioria dos programadores, usar mais do que um *ponteiro para um ponteiro* resultará em confusão considerável, e tornará seus programas muito difíceis de compreender. Por exemplo, o programa a seguir, *ptraptr.c*, usa três níveis de ponteiros para um valor do tipo *int*. Separe um tempo agora para experimentar este programa e desenhe os níveis de indireção em um pedaço de papel até compreender o processamento que *ptraptr* executa:

```
#include <stdio.h>

int qual_e_o_valor(int ***ptr)
 {
   return(***ptr);
 }

void main(void)
 {
   int *nivel_1, **nivel_2, ***nivel_3, valor = 1001;

   nivel_1 = &valor;
   nivel_2 = &nivel_1;
   nivel_3 = &nivel_2;
   printf("O valor é %d\n", qual_e_o_valor(nivel_3));
 }
```

531 COMPREENDENDO AS ESTRUTURAS

Como você aprendeu, uma matriz é uma variável que permite que seus programas armazenem múltiplos valores do mesmo tipo. Em outras palavras, uma matriz permite que seus programas agrupem informações relacionadas em uma única variável, tal como 100 notas de exames ou 50 salários de funcionários. À medida que seus programas se tornarem mais complexos, algumas vezes você desejará agrupar informações relacionadas que têm tipos diferentes. Por exemplo, assuma que você tenha um programa que trabalhe com informações de funcionários. Você poderia precisar controlar as seguintes informações para cada funcionário:

```
char nome[64];
int idade;
char cpf[11];      // Número do CPF
char categoria;
float salario;
unsigned num_func;
```

Assuma que você tenha várias funções diferentes no seu programa que trabalham com informações dos funcionários. Toda vez que seu programa chama uma função, você precisa garantir que tenha especificado todos os parâmetros e na ordem correta. Como discutido na seção Funções, quanto mais parâmetros seus programas passam para as funções, mais difíceis de compreender seus programas serão, e maior a possibilidade de erro. Para reduzir essa complexidade, seus programas podem criar uma *estrutura*, que agrupa as informações relacionadas em uma variável. Por exemplo, a declaração a seguir cria uma estrutura, chamada *Funcionario*, que contém os campos de funcionários mostrados anteriormente:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   char categoria;
```

```
    float salario;
    unsigned num_func;
};
```

Como você aprenderá nas dicas a seguir, essa declaração cria uma estrutura do tipo *Funcionario:*

UMA ESTRUTURA É UM GABARITO PARA A DECLARAÇÃO DE VARIÁVEIS 532

Na dica anterior você aprendeu que C lhe permite agrupar informações relacionadas em uma estrutura. Por si só, a definição de uma estrutura não cria variáveis. Em vez disso, a definição especifica um gabarito que seus programas podem usar mais tarde para declarar variáveis. Portanto, a definição de uma estrutura não aloca memória alguma. Em vez disso o compilador simplesmente toma nota da definição caso seu programa mais tarde declare uma variável do tipo estrutura.

O DESCRITOR DE UMA ESTRUTURA É O NOME DA ESTRUTURA 533

Na Dica 531 vimos que C lhe permite agrupar variáveis relacionadas em uma estrutura. Usando a palavra-chave *struct*, seus programas podem declarar uma estrutura, como mostrado aqui:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   char categoria;
   float salario;
   unsigned num_func;
};
```

No exemplo anterior, o nome da estrutura é *Funcionario.* Os programadores C referenciam o nome da estrutura como o *descritor* (ou *tag*) da estrutura. Como você aprenderá na Dica 534, seus programas podem usar o descritor da estrutura para declarar variáveis de um tipo específico. A declaração a seguir cria uma estrutura chamada *Forma:*

```
struct Forma
 {
   int tipo;   // 0 = círculo, 1 = quadrado, 2 = triângulo
   int cor;
   float raio;
   float area;
   float perimetro;
};
```

DECLARANDO UMA VARIÁVEL ESTRUTURA DE MODOS DIFERENTES 534

Na Dica 531 você aprendeu que C lhe permite agrupar informações relacionadas em uma estrutura. Como você sabe, a definição de uma estrutura por si só não cria uma variável usável. Em vez disso, a definição simplesmente serve como um gabarito para a declaração de futuras variáveis. C fornece dois modos de declarar variáveis de um tipo de estrutura específica. Primeiro, assuma que seu programa declara uma estrutura do tipo *Funcionario,* como mostrado aqui:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   char categoria;
   float salario;
   unsigned num_func;
 };
```

Seguindo a definição da estrutura, seus programas podem declarar variáveis do tipo *Funcionario*, como mostrado aqui:

```
struct Funcionario info_func;
struct Funcionario novo_func, func_demitido;
```

Em seguida, C lhe permite declarar variáveis de um tipo de estrutura seguindo a definição da estrutura, como mostrado aqui:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   char categoria;
   float salario;
   unsigned num_func;
 }info_func, novo_func, func_demitido
```

535 COMPREENDENDO OS MEMBROS DA ESTRUTURA

Você sabe que C lhe permite agrupar informações relacionadas dentro das estruturas. Por exemplo, o comando a seguir cria uma variável chamada *triangulo* usando a estrutura *Forma:*

```
struct Forma
 {
   int tipo;   // 0 = círculo, 1 = quadrado, 2 = triângulo
   int cor;
   float raio;
   float area;
   float perimetro;
 };  triangulo
```

Cada informação na estrutura é um *membro*. No caso da estrutura *Forma*, existem cinco membros: *tipo*, *cor*, *raio*, *area* e *perimetro*. Para acessar um membro específico, você usa o operador ponto (.) de C. Como um exemplo, os comandos a seguir atribuem valores a membros diferentes da variável *triangulo:*

```
triangulo.tipo = 2;
triangulo.perimetro = 30.0;
triangulo.area = 45.0;
```

536 VISUALIZANDO UMA ESTRUTURA

C lhe permite agrupar informações relacionadas em estruturas. Quando você declarar uma variável de um tipo de estrutura específica, C alocará memória suficiente para armazenar os valores para cada membro da estrutura. Por exemplo, se você declarar uma estrutura do tipo *Funcionario*, C alocará memória como mostrado na Figura 536.

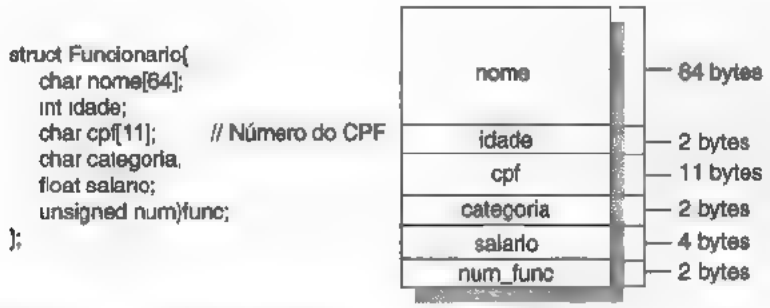

Figura 536 Um mapa lógico da memória que C aloca para armazenar uma estrutura.

PONDO UMA ESTRUTURA EM USO 537

Como você aprendeu, C lhe permite agrupar informações relacionadas em uma estrutura. Na seção Data e Hora, mais à frente, você usará a função *getdate* para determinar a data atual do sistema. A função atribui a data atual aos membros de uma estrutura do tipo *data*, como mostrado aqui:

```
struct data
 {
   int da_ano;     // Ano atual
   char da_dia;    // Dia do mês
   char da_mes;    // Mês do ano
 };
```

O programa a seguir, *dosdata.c*, usa a função *getdate* para atribuir a data à variável *data_atual*:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct date data_atual;

   getdate(&data_atual);
   printf("Data atual: %d-%d-%d\n", data_atual.da_day,
      data_atual.da_mon, data_atual.da_year);
 }
```

Como a função precisa modificar o valor do parâmetro, o programa passa a variável estrutura para a função por referência (por endereço).

PASSANDO UMA ESTRUTURA A UMA FUNÇÃO 538

Como você aprendeu, C lhe permite agrupar informações relacionadas dentro de uma estrutura. Como todas as variáveis, C lhe permite passar variáveis para uma função. O programa a seguir, *estrufun.c*, passa uma estrutura do tipo *Forma* para a função *exibe_estru*, que, por sua vez, exibe os membros da estrutura:

```
#include <stdio.h>

struct Forma
 {
   int tipo;
   int cor;
   float raio;
   float area;
   float perimetro;
 };
```

```
void exibe_estru(struct Forma forma)
 {
   printf("forma.tipo %d\n", forma.tipo);
   printf("forma.cor %d\n", forma.cor);
   printf("forma.raio %f forma.area %f forma.perimetro %f\n",
       forma.raio, forma.area, forma.perimetro);
 }

void main(void)
 {
   struct Forma circulo;

   circulo.tipo = 0;
   circulo.cor = 1;
   circulo.raio = 5.0;
   circulo.area = 22.0 / 7.0 * circulo.raio * circulo.raio;
   circulo.perimetro = 2.0 * 22.0 / 7.0 * circulo.raio;
   exibe_estru(circulo);
 }
```

539 ALTERANDO UMA ESTRUTURA DENTRO DE UMA FUNÇÃO

Na dica anterior você aprendeu que pode passar estruturas para as funções exatamente como pode passar variáveis de todos os tipos. Para alterar os membros de uma estrutura dentro de uma função, você precisa passar a estrutura por endereço (exatamente como passaria uma variável cujo valor quer alterar). O programa a seguir, *mudaest.c*, chama a função *muda_estru*, que altera os valores contidos em uma estrutura do tipo *Forma*:

```
#include <stdio.h>

struct Forma
 {
  int tipo;
  int cor;
  float raio;
  float area;
  float perimetro;
 };

void muda_estru(struct Forma *forma)
 {
   (*forma).tipo = 0;
   (*forma).cor = 1;
   (*forma).raio = 5.0;
   (*forma).area = 22.0 / 7.0 * (*forma).raio * (*forma).raio;
   (*forma).perimetro = 2.0 * 22.0 / 7.0 * (*forma).raio;
 }

void main(void)
 {
   struct Forma circulo;

   muda_estru(&circulo);
   printf("circulo.tipo %d\n", circulo.tipo);
   printf("circulo.cor %d\n", circulo.cor);
   printf("circulo.raio %f circulo.area %f circulo.perimetro %f\n",
      circulo.raio, circulo.area, circulo.perimetro);
 }
```

Para modificar os membros de uma estrutura, o programa passa para a função um ponteiro para a estrutura. Dentro da função, os comandos desreferenciam os membros do ponteiro usando o operador *asterisco de indireção:*

```
(*ponteiro).membro = valor;
```

COMPREENDENDO A INDIREÇÃO (*PONTEIRO).MEMBRO 540

Para alterar o membro de uma estrutura dentro de uma função, o programa precisa passar um ponteiro para a estrutura. Dentro da função, os comandos desreferenciam o ponteiro usando o operador *asterisco de indireção,* como mostrado aqui:

```
(*ponteiro).membro = valor;
```

Para resolver o ponteiro, C inicia dentro dos parênteses, obtendo primeiro a localização da estrutura. Em seguida, C acrescenta ao endereço o deslocamento do membro especificado. Se você omitir os parênteses, C assume que o membro é um ponteiro e usa o operador asterisco de indireção para resolvê-lo, como mostrado aqui:

```
*ponteiro.membro = valor;
```

A sintaxe dos parênteses omitidos seria correta para uma estrutura tendo um membro que fosse um ponteiro, tal como o seguinte:

```
struct Planeta
 {
   char nome[48];
   int *algum_ponteiro;
 } planeta;
```

Como você pode ver, o segundo membro é um ponteiro para um valor do tipo *int.* Assumindo que o programa anteriormente atribuiu o ponteiro a uma posição de memória, o seguinte comando coloca o valor 5 na posição de memória:

```
*planeta.algum_ponteiro = 5;
```

USANDO O FORMATO PONTEIRO—>MEMBRO 541

Na dica anterior vimos que, para alterar o membro de uma estrutura dentro de uma função, o programa precisa passar um ponteiro para a estrutura. Para desreferenciar o ponteiro dentro da função, C fornece dois formatos. Primeiro, como você viu, pode referenciar o membro de uma estrutura da seguinte forma:

```
(*ponteiro).membro = valor;
algum_valor = (*ponteiro).membro;
```

Segundo, C lhe permite usar o seguinte formato:

```
ponteiro->membro = valor;
algum_valor = ponteiro->membro;
```

O programa a seguir, *mudamemb.c,* usa o seguinte formato dentro da função *muda_estru* para referenciar os membros de uma estrutura passada para a função por endereço:

```
#include <stdio.h>

struct Forma
 {
   int tipo;
   int cor;
   float raio;
   float area;
   float perimetro;
 };
```

```
void muda_estru(struct Forma *forma)
 {
   forma->tipo = 0;
   forma->cor = 1;
   forma->raio = 5.0;
   forma->area = 22.0 / 7.0 * forma->raio * forma->raio;
   forma->perimetro = 2.0 * 22.0 / 7.0 * forma->raio;
 }

void main(void)
 {
   struct Forma circulo;

   muda_estru(&circulo);
   printf("circulo.tipo %d\n", circulo.tipo);
   printf("circulo.cor %d\n", circulo.cor);
   printf("circulo.raio %f circulo.area %f circulo.perimetro %f\n",
      circulo.raio, circulo.area, circulo.perimetro);
 }
```

542 USANDO UMA ESTRUTURA SEM NOME

Como você aprendeu, o *descritor* é o nome da estrutura. Usando o descritor, seus programas podem declarar variáveis de um tipo de estrutura específica. No entanto, quando você declara variáveis de tipos de estrutura imediatamente seguindo a definição da estrutura, você não tem necessariamente que especificar o nome da estrutura. Por exemplo, a declaração a seguir cria duas variáveis de uma estrutura:

```
struct
 {
   int tipo;       // 0 = círculo, 1 = quadrado, 2 = triângulo
   int cor;
   float raio;
   float area;
   float perimetro;
 } triangulo, circulo;
```

Caso seu programa não venha a referenciar mais tarde uma estrutura por nome (tal como um protótipo de função ou parâmetros formais), então você poderá omitir o nome da estrutura, como acaba de ser mostrado. No entanto, quando você inclui o descritor, fornece aos outros programadores que lêem seu programa uma explicação sobre o propósito da estrutura. Quando você incluir nomes de descritores representativos, tornará seus programas mais legíveis.

543 COMPREENDENDO O ESCOPO DE DEFINIÇÃO DE UMA ESTRUTURA

Na seção Funções, você aprendeu que *escopo* define a região do programa dentro da qual um identificador (tal como uma variável ou função) é conhecido. Quando você define uma estrutura, precisa considerar o escopo da estrutura. Se você examinar os programas anteriores que trabalham com estrutura dentro de funções, verá que o programa define a estrutura fora e antes das funções que a usam. Como resultado, as definições das estruturas têm um escopo global, o que permite que todas as funções seguintes as referenciem. Se, em vez disso, o programa tivesse definido as estruturas dentro de *main*, a única função que teria conhecimento da existência da estrutura seria *main*. Se você precisar que várias funções do seu programa usem a definição de uma estrutura, deverá definir a estrutura fora das suas funções em algum ponto antes de todas as funções que precisam acessar a estrutura.

INICIALIZANDO UMA ESTRUTURA 544

Já vimos que C lhe permite inicializar as matrizes na declaração. De um modo similar, seus programas também podem inicializar uma estrutura na declaração. O programa a seguir, *inicestr.c*, declara e inicializa uma estrutura do tipo *Forma:*

```
#include <stdio.h>

void main(void)
 {
   struct Forma
   {
      int tipo;
      int cor;
      float raio;
      float area;
      float perim;
   } circulo = {0, 1, 5.0, 78.37, 31.42};

   printf("circulo.tipo %d\n", circulo.tipo);
   printf("circulo.cor %d\n", circulo.cor);
   printf("circulo.raio %f circulo.area %f circulo.perim f\n",
      circulo.raio, circulo.area, circulo.perim);
 }
```

Como o programa somente usa a estrutura dentro de *main*, o programa define a estrutura dentro de *main*.

EFETUANDO E/S EM ESTRUTURAS 545

Várias dicas apresentadas nesta seção usaram *printf* para exibir o valor de um ou mais membros da estrutura. Quando você efetua operações de E/S na tela ou no teclado que afetam os membros da estrutura, precisa efetuar E/S um membro de cada vez. No entanto, ao ler ou gravar estruturas de/ou para um arquivo, seus programas podem trabalhar com a estrutura inteira. Se seu programa usa canais de arquivos, você pode usar as funções *fwrite* e *fread* para gravar e ler estruturas. A seção Arquivos, Diretórios e Discos ilustra como usar *fwrite* e *fread* para efetuar E/S de estruturas. Para compreender melhor esse processo, referencie os programas *dhsai.c* e *dhent.c* localizados dentro do diretório Dica0422. Caso seus programas usem indicativos de arquivos, você poderá usar as funções *read* e *write* para efetuar E/S em estruturas. Os arquivos *dhsai.c* e *dhent.c* ilustram como seus programas podem usar *write* e *read* para efetuar E/S em estruturas. Cada uma das funções de E/S que acabam de ser descritas lêem ou gravam um intervalo de bytes. Quando C armazena uma estrutura na memória, a estrutura é na verdade apenas um intervalo de bytes. Portanto, para usar uma estrutura com essas funções, simplesmente passe um ponteiro para a estrutura, como mostrado nos programas de exemplo.

USANDO UMA ESTRUTURA DENTRO DE OUTRA 546

Como você aprendeu, C lhe permite armazenar informações relacionadas dentro de estruturas. Dentro de uma estrutura, você pode incluir membros de qualquer tipo (*int*, *float* etc;), bem como os membros que são eles mesmos estruturas. Por exemplo, a seguinte declaração de estrutura inclui uma estrutura do tipo *Data*, que contém a data de admissão de um funcionário:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   struct Data
    {
      int dia;
      int mes;
      int ano;
```

```
    } admissao;
    char categoria;
    float salario;
    unsigned num_func;
  } novo_func;
```

Para acessar um membro de uma estrutura que está dentro de outra, você usa o operador *ponto*, primeiro para especificar a estrutura embutida, e, depois, para especificar o membro desejado, como mostrado aqui:

```
novo_func.admissao.mes = 12;
```

547 ESTRUTURAS QUE CONTÊM MATRIZES

Os membros da estrutura podem ser de qualquer tipo, incluindo estruturas ou matrizes. Quando um membro da estrutura é uma matriz, seus programas referenciam o membro da matriz como fariam com qualquer matriz, exceto que o nome da variável e o operador ponto precedem o nome da matriz. Por exemplo, o programa a seguir, *estrumtz.c*, inicializa vários campos de estrutura, incluindo uma matriz. O programa então percorre em um laço os elementos da matriz, exibindo seus valores:

```
#include <stdio.h>

void main(void)
 {
   struct Date
    {
      char mes_nome[64];
      int  mes;
      int  dia;
      int  ano;
    } data_atual = { "Setembro", 9, 7, 1998 };
   int i;

   for (i = 0; data_atual.mes_nome[i]; i++)
     putchar(data_atual.mes_nome[i]);
 }
```

548 CRIANDO UMA MATRIZ DE ESTRUTURAS

Como você aprendeu, uma matriz permite que seus programas armazenem múltiplos valores do mesmo tipo. A maioria das matrizes apresentadas nesta seção foi do tipo *int*, *float*, ou *char*. No entanto, C também lhe permite declarar matrizes de um tipo de estrutura específico Por exemplo, a declaração a seguir cria uma matriz capaz de armazenar informações sobre 100 funcionários:

```
struct Funcionario
 {
   char nome[64];
   int idade;
   char cpf[11];      // Número do CPF
   char categoria;
   float salario;
   unsigned num_func;
 } equipe[100];
```

Assumindo que o programa tenha atribuído valores a cada funcionário, o seguinte laço *for* exibirá o nome e o número de cada funcionário:

```
for (emp = 0; emp < 100; emp++)
   printf("Funcionário: %s   Número: %d\n",
equipe[emp].nome, equipe[emp].num_func);
```

Quando você usar uma matriz de estruturas, simplesmente acrescente o operador ponto a cada elemento da matriz.

COMPREENDENDO OS SERVIÇOS DO SISTEMA DO DOS 549

Sabemos que, o DOS é o sistema operacional para IBM-PC e compatíveis. O DOS lhe permite rodar programas e armazenar informações no disco. Além disso, o DOS fornece serviços que permitem que os programas aloquem memória, acessem dispositivos, tal como a impressora, e gerencie outros recursos do sistema. Para ajudar seus programas a aproveitar as capacidades existentes no DOS — tais como determinar a quantidade de espaço livre no disco, criar ou selecionar um diretório, ou até capturar as digitações — o DOS fornece um conjunto de serviços que seus programas podem usar. Ao contrário das funções que as bibliotecas de execução de C fornecem, seus programas não acessam os serviços do DOS usando uma interface simples de chamada de função. Em vez disso, os programadores escreviam os serviços para que outros programadores pudessem acessar os serviços no nível de linguagem Assembly, usando registradores e interrupções. No entanto, como você aprenderá nesta seção, C na verdade facilita para seus programas fazer uso dos serviços do DOS sem forçá-lo a usar a linguagem Assembly. Além disso, a biblioteca de execução de C frequentemente fornece uma interface a muitos dos serviços do DOS por meio de uma função.

Quando você cria seu programa, algumas vezes tem a escolha de usar uma função da biblioteca de execução de C ou um serviço do DOS. Como regra, você deverá usar as funções da biblioteca de execução sempre que possível, em vez dos serviços do DOS, porque usar as funções da biblioteca de execução aumentará a portabilidade do seu programa. Em alguns casos, as funções da biblioteca de execução de C fornecidas no UNIX ou ambiente Windows correspondem àquelas que seu compilador do DOS fornece. Quando você usar a função de biblioteca de execução de C, em vez do serviço do DOS — não será preciso alterar seu programa para ele rodar sob o UNIX. Em vez disso, você simplesmente precisar recompilar.

Quando você programar para o Windows, usará um conjunto diferente de serviços chamado serviços do sistema Windows (também conhecidos como Interface de Programação de Aplicativos (API) do Windows). A API lhe permite controlar a maioria dos serviços dentro do Windows, incluindo serviços de arquivo, serviços de memória, e assim por diante, usando um conjunto de funções de C++. Seus programas podem chamar as funções de C++ a partir de dentro de seu código normal, e retornar valores para seus programas de forma muito parecida aos que fazem as chamadas de serviços do DOS. Você aprenderá mais sobre os serviços do sistema Windows nas Dicas 1251 a 1500.

COMPREENDENDO OS SERVIÇOS DA BIOS 550

BIOS significa Serviços Básicos de Entrada/Saída. Em resumo, BIOS é um circuito integrado dentro do seu computador que contém as instruções que o computador usa para escrever na tela ou na impressora, para ler caracteres do teclado, ou para ler e gravar no seu disco. Como foi o caso com os serviços do DOS, seus programas podem usar os serviços da BIOS para executar diferentes operações. Por exemplo, você usa um serviço da BIOS para determinar o número de portas paralelas ou seriais, o tipo de monitor de vídeo, ou o número de unidades de disco disponíveis. Como os serviços do DOS, as rotinas da BIOS foram originalmente projetadas para uso nos programas em linguagem Assembly. No entanto, a maioria dos compiladores C fornece funções da biblioteca *de execução que permitem que seus programas usem esses serviços sem a necessidade da linguagem Assembly.* Muitos programadores confundem os serviços do DOS e da BIOS. Como mostra a Figura 550.1, BIOS reside diretamente acima do hardware do seu computador. Os serviços do DOS estão em um nível acima da BIOS, e os seus programas estão em um nível acima do DOS.

Figura 550.1 O relacionamento entre BIOS, DOS e programas.

No entanto, algumas vezes, o DOS e até mesmo seus programas podem ignorar a BIOS e acessar diretamente o hardware. Por exemplo, um aplicativo que precise fornecer vídeo muito rápido, poderia ignorar o DOS e BIOS e trabalhar diretamente na memória de vídeo. No entanto, como regra, somente os programadores experientes devem ignorar o DOS e a BIOS. O DOS e BIOS fazem uma considerável verificação de erro, o que simplifica as tarefas do programador.

Todas as variedades de Windows, incluindo o Windows 95 e o Windows NT, chamam seus próprios serviços do sistema. No entanto, exatamente como com os serviços do sistema DOS, os serviços do Windows eventualmente chamam os serviços da BIOS para acessar o hardware do computador. No entanto, embora ainda seja possível ignorar os serviços do sistema Windows e chamar os serviços da BIOS diretamente, em geral não é uma boa idéia fazer isso, devido ao projeto do sistema operacional Windows. Tipicamente, você tem os melhores resultados dentro dos programas Windows usando a API do Windows, e apenas deverá chamar os serviços da BIOS se esses forem absolutamente necessários. A Figura 550.2 mostra o relacionamento entre BIOS, DOS, Windows e programas.

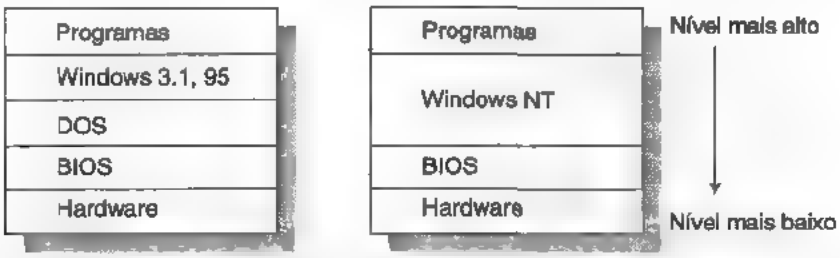

Figura 550.2 O relacionamento entre BIOS, DOS, Windows e programas.

551 COMPREENDENDO OS REGISTRADORES

Quando seu programa é executado, ele precisa residir na memória do seu computador. A unidade central de processamento (CPU) do seu computador lerá as instruções e dados de que seu programa precisa da memória. Para aumentar o desempenho, a CPU contém várias posições de armazenamento temporário denominadas *registradores*. Como esses registradores residem dentro da própria CPU, esta pode acessar o conteúdo de cada registrador muito rapidamente. Em geral, a CPU usa quatro tipos de registradores: *segmento*, *deslocamento*, *propósito geral* e *sinalizadores* (ou *flags*). A Tabela 551 descreve rapidamente o uso de cada tipo de registrador.

Tabela 551 Os tipos de registradores do PC.

Tipo de Registrador	Uso
Segmento	Contém o endereço inicial de um bloco de memória, tal como o início do código ou os dados do seu programa
Deslocamento	Contém o deslocamento de 16 bits dentro de um bloco de memória, tal como a localização de uma variável específica dentro do segmento de dados do seu programa
Propósito Geral	Armazena temporariamente os dados do programa
Sinalizadores	Contém informações de status e de erro do processador

O PC usa um valor de segmento e um valor de deslocamento para localizar itens na memória. Quando você usa os serviços do DOS, pode querer atribuir os endereços de segmento e de deslocamento de uma ou mais variáveis para diferentes registradores de segmento. O PC fornece quatro registradores de propósito geral, chamados *AX*, *BX*, *CX* e *DX*. Cada registrador de propósito geral pode conter 16 bits de dados (2 bytes). Em alguns casos, você poderia querer armazenar somente um byte de informações dentro de um registrador. Para lhe ajudar a fazer isso, o PC lhe permite acessar os bytes superior e inferior de cada registrador usando os nomes mostrados na Figura 511.1.

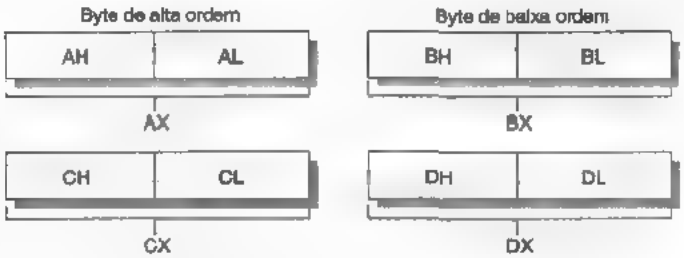

Figura 551.1 Os quatro registradores de propósito geral do PC.

Ao usar os serviços do DOS e da BIOS, você colocará os parâmetros para qualquer serviço que seu programa estiver prestes a chamar dentro de registradores de propósito geral. Quando o serviço terminar, o DOS ou a BIOS poderão colocar o resultado do serviço em um dos registradores de propósito geral. Finalmente, o registrador de flags armazena o estado da CPU e possíveis valores de status de erro. Quando os serviços do DOS e da BIOS terminam, geralmente eles ligam ou desligam diferentes bits dentro do registrador de flags para indicar sucesso ou um erro. A Figura 551.2 ilustra os bits dentro do registrador de flags (os retângulos cinza representam os bits não utilizados).

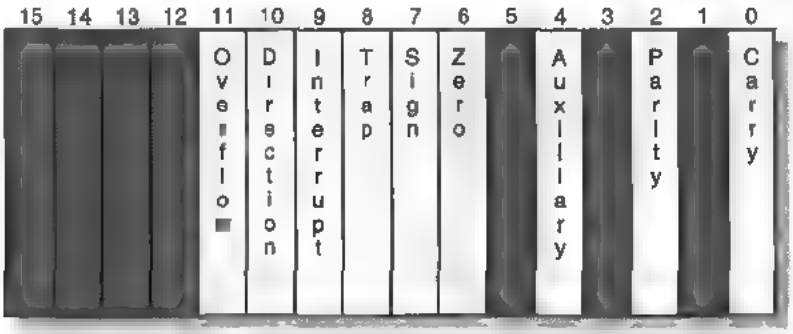

Figura 551.2 Os bits dento do registrador de flags do PC.

Várias dicas apresentadas nesta seção discutem os registradores de segmento e de deslocamento em maiores detalhes.

COMPREENDENDO O REGISTRADOR DE FLAGS 552

Como você aprendeu, o registrador de flags contém informações de status e de erro da CPU. Após o PC completar diferentes operações, tais como adicionar, subtrair ou comparar, ele define certos bits dentro do registrador de flags. Da mesma forma, muitos serviços do DOS e da BIOS ligam o sinalizador Carry para indicar um erro. A Tabela 552 descreve os bits que os serviços da BIOS e do DOS usam dentro do registrador de flags.

Tabela 552 Bits dentro do registrador de flags.

Bit	Sinalizador	Significado
0	Carry	Indica um "vai um" matemático
2	Parity	Indica que uma operação aritmética resultou em um número par de bits ligados
4	Auxiliary	Indica que um ajuste é necessário após uma operação aritmética em BCD (código binário decimal)
6	Zero	Indica um resultado zero de uma operação de comparação ou aritmética
7	Sign	Indica um resultado negativo
8	Trap	Usada para interceptação do depurador
10	Direction	Controla a direção das instruções string
11	Overflow	Indica um extravasamento aritmético

Quando você usa um serviço do DOS ou da BIOS dentro do seu programa, certifique-se de que seu programa testa o bit de flag que o serviço define para determinar se houve sucesso ou não.

COMPREENDENDO AS INTERRUPÇÕES DE SOFTWARE 553

Uma *interrupção* ocorre quando a CPU precisa parar temporariamente o que está fazendo para poder efetuar uma operação diferente. Quando a operação termina, a CPU pode retomar seu trabalho original como se nunca tivesse parado. Existem dois tipos de interrupções: de hardware e de software. Os dispositivos conectados com o seu computador, tais como o relógio, a unidade de disco, ou teclado, causam interrupções. Quando os desenvolvedores originais projetaram o PC, eles forneceram suporte para até 256 interrupções, numeradas de 0 a 255. Como o hardware do seu computador precisa somente de um pequeno número dessas interrupções, muitas estão disponíveis para o software usar. Os serviços da BIOS, por exemplo, fazem uso das interrupções de 5 e 10H até

1FH (16 decimal até 31 decimal). Da mesma forma, o DOS usa as interrupções 21H até 28H (33 decimal até 40 decimal) e 2FH (47 decimal).

Quando você escreve programas em linguagem Assembly, atribui parâmetros aos registradores do PC, e, depois, chama a interrupção que corresponde ao serviço do sistema que desejar. Por exemplo, a BIOS usa a interrupção 10H para acessar o vídeo. Assim, para exibir uma letra na tela, você atribui a letra que quer para o registrador AL, atribui o valor 9 ao registrador AH (o que instrui a BIOS a executar uma operação de escrita no vídeo), atribui o atributo que quer (negrito, piscante, normal etc.) ao registrador BX, e, depois, chama a INT 10H, como mostrado aqui:

```
MOV AL,41    ; A é o ASCII 41H
MOV AH,9     ; Solicita escrita no vídeo
MOV BX,7     ; Atributo do caractere
MOV CX,1     ; Número de caracteres a escrever
INT 10       ; Efetua o serviço do vídeo
```

Como você aprenderá, a maioria dos serviços do DOS usa a INT 21H. Felizmente, você não precisa trabalhar com linguagem Assembly dentro de seus programas C para chamar um serviço.

Nota: Embora seja possível usar interrupções de software dentro de seus programas Windows, você deverá executar a maioria das atividades dentro de seus programas Windows usando a API do Windows.

554 Usando a BIOS para Acessar a Impressora

Várias dicas apresentadas neste livro escreveram saída na impressora usando o indicativo de arquivo *stdprn*. No entanto, antes que seus programas possam efetuar E/S na impressora, você pode querer que o programa verifique se a impressora está ligada e se tem papel. Para fazer isso, seus programas poderão usar a função *biosprint*. Você implementará a função *biosprint* como mostrado aqui:

```
#include <bios.h>

int biosprint(int comando, int byte, int num_porta);
```

O parâmetro *comando* especifica uma das operações listadas na Tabela 554.1.

Tabela 554.1 Os valores possíveis para o parâmetro ***comando***.

Comando	Significado
0	Imprime o byte especificado
1	Inicializa a porta da impressora
2	Lê o status da impressora

Se você está imprimindo um caractere, *byte* especifica o valor ASCII ou ASCII estendido ou o caractere que você quer. O parâmetro *num_porta* especifica a porta da impressora na qual você quer imprimir, onde 0 é LPT1, 1 é LPT2, e assim por diante

A função *biosprint* retorna um valor inteiro no intervalo de 0 até 255, cujos bits estão definidos na Tabela 554.2.

Tabela 554.2 Os bits de status que ***biosprint*** retorna.

Bit	Significado Se Ligado
0	Tempo limite do dispositivo
3	Erro de E/S
4	Impressora selecionada
5	Falta de papel
6	Reconhecimento do dispositivo
7	Dispositivo não pronto

O programa a seguir, *checaimp.c*, usa a função *biosprint* para testar o status da sua impressora repetidamente até que você pressione uma tecla qualquer. Rode o programa a seguir e experimente com a sua impressora, colocando-a fora de linha, removendo o papel e assim por diante. Quando você fizer isso, o programa deverá exibir diferentes mensagens na sua tela, como mostrado aqui:

```
#include <bios.h>
```

```
#include <conio.h>
#include <stdio.h>

void main(void)
  {
   int status = 0;
   int status_antigo = 0;

   do
     {
       status = biosprint(2, 0, 0);  // Lê LPT1
       if (status != status_antigo)
         {
         if (status & 1)
            printf ("Tempo-Limite\t");
         if (status & 8)
            printf ("Erro na Saída\t");
         if (status & 16)
            printf ("Impressora Selecionada\t");
         if (status & 32)
            printf ("Sem Papel\t");
         if (status & 64)
            printf ("Reconhecimento\t");
         if (status & 128)
            printf ("Impressora Não-Ocupada");
         printf ("\n");
         status_antigo = status;
       }
     }
   while (! kbhit());
 }
```

Nota: Muitos compiladores fornecem uma função chamada ***_bios_printer*** *que é similar a* ***biosprint****. Para acessar uma impressora conectada a uma porta serial, você precisa usar a função* ***_bios_serialcom****.*

Nota: O CD-ROM que acompanha este livro inclui o programa ***win_print.cpp****, que usa a API do Windows para enviar informações para uma impressora.*

INFORMAÇÃO DE CTRL+BREAK 555

Quando você trabalha dentro do ambiente do DOS, o comando DOS BREAK lhe permite habilitar e desabilitar a verificação estendida de Ctrl+Break. Quando você habilita a verificação estendida, o DOS aumenta o número de operações após as quais ele verifica um Ctrl+C ou um Ctrl+Break pressionado pelo usuário. Quando você desabilita a verificação estendida de Ctrl+Break, o DOS somente verifica um Ctrl+Break após efetuar E/S na tela, teclado ou impressora. Muitos compiladores C fornecem duas funções, *getcbrk* e *setcbrk*, que seus programas podem usar para obter e definir o estado da verificação Ctrl+Break. Você implementará *getcbrk* e *setcbrk*, como mostrado aqui:

```
#include <dos.h>

int getcbrk(void);
int setcbrk(int definição);
```

A função *getcbrk* retorna o valor 0 se você tiver desabilitado a verificação de Ctrl+Break e 1 se ela estiver ativa. Da mesma forma, a função *setcbrk* usa os valores 0 e 1, respectivamente, para desabilitar e habilitar a verificação estendida. A função *setcbrk* também retorna o valor 0 ou 1, dependendo do estado da verificação estendida que você selecionou. O programa a seguir, *ctrlbrk.c*, usa a função *setcbrk* para desabilitar a verificação estendida do Ctrl+Break. O programa usa o valor de retorno da função *getcbrk* para exibir a definição anterior, como mostrado aqui:

```
#include <stdio.h>
#include <dos.h>

void main(void)
```

```
{
  printf("Status estendido anterior de Ctrl-Break %s\n",
     (getcbrk()) ? "Ligado": "Desligado");
  setcbrk(0);     // Desliga
}
```

*Nota: A função **setcbrk** define o estado da verificação Ctrl+Break para o sistema, não apenas para o programa atual. Quando o programa termina, o estado selecionado anteriormente permanece em efeito. Lembre-se, você tratará as mensagens dentro do Windows usando comandos diferentes; e interceptar Ctrl+Break geralmente não será útil dentro de programas Windows.*

556 COMPREENDENDO POSSÍVEIS EFEITOS COLATERAIS DO DOS

Na dica anterior você aprendeu como usar a função *setcbrk* para modificar o estado da verificação Ctrl+Break estendida. Da mesma forma, na seção Arquivos, Diretórios e Discos você aprendeu como alterar o estado da verificação do disco. Várias outras dicas apresentaram modos de seus programas alterarem a unidade ou o diretório atual. Quando seus programas efetuam essas operações, eles devem salvar as definições originais para poder restaurá-las antes de terminar.

A não ser que o programa tenha o propósito explícito de alterar uma ou mais definições, ele não deverá deixar a definição alterada após terminar. Essas alterações nas definições são chamadas *efeitos colaterais*, que você deverá evitar. Por exemplo, quando um usuário roda um programa de orçamento, a unidade e o diretório padrões do usuário não devem mudar após o programa terminar. Da mesma forma, mudanças mais sutis, tais como a verificação da desabilitação do disco ou a verificação estendida de Ctrl+Break, não devem ocorrer. À medida que você for criando um programa, inclua os comandos adicionais que seu sistema irá requerer para restaurar as definições originais do ambiente.

Nota: Lembre-se, você tratará mensagens dentro do Windows usando diferentes comandos. Interceptar Ctrl+Break geralmente não será útil dentro dos programas Windows.

557 SUSPENDENDO UM PROGRAMA TEMPORARIAMENTE

Na seção Data e Hora, mais à frente você usará a função *delay* para provocar uma pausa no programa por um número específico de milissegundos. De um modo similar, seus programas podem usar a função *sleep* para especificar o intervalo de pausa em segundos, como mostrado aqui:

```
#include <dos.h>

void sleep(unsigned segundos);
```

Como a função *delay* trabalha com milissegundos, ela é mais exata que *sleep*. No entanto, você pode usar a função *sleep* para aumentar a portabilidade para outros sistemas operacionais. A maioria dos sistemas operacionais fornece uma função *sleep*, que permite que os programas fiquem em um estado inativo até um intervalo de tempo transcorrer ou um evento específico ocorrer. O programa a seguir, *sleep_5.c*, usa a função *sleep* para fazer uma pausa por 5 segundos:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   printf("Prestes a adormecer por 5 segundos\n");
   sleep(5);
   printf("Acordar\n");
 }
```

*Nota: Não confunda o comando **sleep** de C com o comando **sleep** da API do Windows. Você aprenderá mais sobre o comando **sleep** da API do Windows na seção mais à frente intitulada Processos e Encadeamentos.*

558 DIVERTINDO-SE COM SOM

Dentro de quase todo PC há um pequeno alto-falante (de baixa qualidade) que os programas normalmente usam para gerar avisos sonoros. No entanto, usando a função *sound* que muitos compiladores C fornecem, seus programas podem gerar sons que emitem diferentes freqüências por meio do alto-falante. A função *sound* permite

que seus programas ativem o alto-falante para emitir um som de freqüência específica. A função *nosound* desativa o alto-falante, como mostrado aqui:

```
#include <dos.h>
void sound(unsigned frequencia);
void nosound(void);
```

O programa a seguir, *sirene.c*, usa a função *sound* para gerar um som similar a uma sirene. Quando você pressiona uma tecla qualquer, o programa desativa o alto-falante usando a função *nosound*, como mostrado aqui:

```
#include <dos.h>
#include <conio.h>

void main(void)
  {
  unsigned freq;
  do
    {
     for (freq = 500; freq <= 1000; freq += 50)
      {
        sound(freq);
        delay(50);
      }
     for (freq = 1000; freq >= 500; freq -= 50)
      {
        sound(freq);
        delay(50);
      }
    }
  while (! kbhit());
  nosound();
  }
```

OBTENDO INFORMAÇÕES ESPECÍFICAS DO PAÍS 559

Como você sabe, o sistema operacional DOS é usado em todo o mundo. Para suportar os usuários internacionais, o DOS suporta diferentes gabaritos de teclado, páginas de código e informações específicas do país. Para ajudar seus programas a determinar as definições atuais do país, seus programas podem usar a função *country*, como mostrado aqui:

```
#include <dos.h>

struct COUNTRY *country(int codigo, struct COUNTRY *info);
```

Se for bem-sucedida, a função retornará um ponteiro para uma estrutura do tipo *COUNTRY*, como mostrado aqui:

```
struct COUNTRY
 {
   int co_date;          // Formato da data
   char co_curr[5];      // Símbolo da moeda
   char co_thsep[2];     // Separador de milhares
   char co_desep[2];     // Separador de decimais
   char co_dtsep[2];     // Separador de data
   char co_tmsep[2];     // Separador de hora
   char co_currstyle;    // Estilo da moeda
   char co_digits;       // Dígitos significativos da moeda
   char co_time;         // Formato da hora
   char co_case;         // Ponteiro para case map
   char co_dasep;        // Separador de data
   char co_fill[10];     // Espaço de preenchimento
 };
```

O valor *codigo* especifica um código de país que você quer selecionar. Se o valor do parâmetro *info* for -1, a função *country* definirá o código de país atual com o código que você especificar. Se o valor de *info* não for -1,

a função *country* atribuirá ao buffer as definições para o código atual de país. O programa a seguir, *pais.c*, exibe as definições atuais de país:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
   struct COUNTRY info;

   country(0, &info);
   if (info.co_date == 0)
      printf("Formato da data: mm/dd/aa\n");
   else if (info.co_date == 1)
      printf("Formato da data: dd/mm/aa\n");
   else if (info.co_date == 2)
      printf("Formato da data: aa/mm/dd\n");
   printf("Símbolo da moeda %s\n", info.co_curr);
   printf("Separador decimal %s\n", info.co_thsep);
   printf("Separador de data %s Separador de hora %s\n",
      info.co_dtsep, info.co_tmsep);
   if (info.co_currstyle == 0)
      printf("Símbolo da moeda precede sem espaço preliminar\n");
   else if (info.co_currstyle == 1)
      printf("Símbolo da moeda segue sem espaços\n");
   else if (info.co_currstyle == 2)
      printf("Símbolo da moeda precede com espaço preliminar\n");
   if (info.co_currstyle == 4)
      printf("Símbolo da moeda segue com espaço\n");
   printf("Dígitos significativos da moeda %d\n",
      info.co_digits);
   if (info.co_time)
      printf("Horário em 24 horas \n");
   else
      printf("Horário em 12 horas\n");
   printf("Separador de data %s\n", info.co_dasep);
  }
```

Nota: O CD-ROM que acompanha este livro inclui o programa win_country.cpp, que retorna as definições atuais de país dentro do Windows.

560 COMPREENDENDO O ENDEREÇO DE TRANSFERÊNCIA DO DISCO

Antes do DOS 3.0, os programas executavam as operações em arquivo usando os blocos de controle de arquivo (FCBs). Por padrão, quando o DOS lia ou gravava informações, ele fazia isso por meio de uma área de memória chamada *área de transferência do disco*. Por padrão, o tamanho da área de transferência do disco era de 128 bytes. O endereço do primeiro byte da área é chamado *área de transferência do disco* (DTA). Por padrão, o DOS usava o deslocamento 80H do prefixo do segmento do programa como o endereço de transferência do disco. Como você aprenderá mais à frente na seção Redirecionando a E/S e Processando Linhas de Comando, o deslocamento 80H do prefixo de segmento do programa também contém a linha de comando do programa. Como a maioria dos programas não usa operações de disco de blocos de controle de arquivo, muitos programadores assumem que podem ignorar o endereço de transferência do disco. Infelizmente, rotinas, tais como *findnext* e *findfirst*, discutidas na seção Arquivos, Diretórios e Discos deste livro, colocam suas rotinas no endereço de transferência do disco, sobrescrevendo a linha de comando do seu programa. Para evitar as operações que usam o endereço de transferência do disco de sobrescrever a linha de comando do seu programa, muitos programadores usam um serviço do DOS para definir o endereço de transferência do disco para apontar para um buffer de memória diferente. Como você aprenderá na dica 561, seus programas podem alterar e determinar o endereço de transferência do disco usando funções da biblioteca de execução.

ACESSANDO E CONTROLANDO A ÁREA DE TRANSFERÊNCIA DO DISCO 561

Na dica anterior você aprendeu que a área de transferência do disco é uma região de 128 bytes que o DOS usa para os serviços de E/S com base em blocos de controle de arquivo ou operações *findfirst* e *findnext*. Para lhe ajudar a controlar a área de transferência do disco, a maioria dos compiladores suporta as funções *getdta* e *setdta*, como mostrado aqui:

```
#include <dos.h>

char *far getdta(void);
void setdta(char far *ender_transferencia_disco);
```

A função *getdta* retorna um ponteiro *far* (32 bits) para a área de transferência do disco atual. Da mesma forma, a função *setdta* lhe permite atribuir o endereço de transferência do disco ao endereço *far* que você especificar. O programa a seguir, *dta.c*, ilustra o uso das funções *getdta* e *setdta*:

```
#include <stdio.h>
#include <dos.h>
#include <malloc.h>

void main(void)
  {
   char far *dta;

   dta = getdta();
   printf("A DTA atual é %lX\n", dta);
   if (MK_FP(_psp, 0x80) == dta)
     printf("A DTA está na mesma posição que a linha de comando\n");
   dta = _fmalloc(128);
   setdta(dta);
   printf("Nova DTA é %lX\n", getdta());
 }
```

Nota: Quando você programa dentro do Windows, não precisa controlar o endereço de transferência do disco, já que o Windows usa o modelo de memória virtual - detalhado dentro da seção Gerenciamento da Memória no Windows — para tratar a maior parte da E/S em arquivo.

USANDO OS SERVIÇOS DE TECLADO DA BIOS 562

O DOS, a BIOS e a biblioteca de execução de C fornecem serviços que permitem que seus programas acessem o teclado. Como regra, você primeiro deve tentar usar as funções da biblioteca de execução de C. Se uma função da biblioteca de execução não funcionar, então use a função do DOS. Se a função não funcionar, experimente os serviços da BIOS. O uso das funções da biblioteca de execução permite que seus programas permaneçam mais portáteis. Para ajudar seus programas a acessar os serviços de teclado da BIOS, a biblioteca de execução de C oferece a função *_bios_keybrd*, como mostrado aqui:

```
#include <bios.h>

unsigned _bios_keybrd(unsigned comando);
```

O parâmetro *comando* especifica a operação desejada. A Tabela 562 lista os valores possíveis que você pode passar para o parâmetro *comando*:

Tabela 562 Os valores possíveis para o parâmetro ***comando***.

Valor	Significado
_KEYBRD_READ	Instrui *_bios_keybrd* a ler um caractere do buffer do teclado. Se o byte menos significativo do valor de retorno é 0, o byte mais significativo contém um código de teclado estendido.
_KEYBRO_READY	Instrui *_bios_keybrd* a determinar se um caractere está presente no buffer do teclado. Se *_bios_keybrd* retorna 0, nenhuma digitação está presente. Se o valor de retorno é 0xFFFF, o usuário pressionou Ctrl+C.
_KEYBRD_SHIFTSTATUS	Instrui *_bios_keybrd* a retornar o estado de modificação do teclado: Bit 7 Ligado — INS está ligado Bit 6 Ligado — Capslock está ligado Bit 5 Ligado — Numlock está ligado Bit 4 Ligado — ScrollLock está ligado Bit 3 Ligado — Tecla Alt pressionada Bit 2 Ligado — Tecla Ctrl pressionada Bit 1 Ligado — Tecla Shift esquerda pressionada Bit 0 Ligado — Tecla Shift direita pressionada
_NKEYBRD_READ	Instrui *bios_keybrd* a ler um caractere do buffer de teclado. Se o byte menos significativo do valor de retorno é 0, o byte mais significativo contém um código de teclado estendido. _NKEYBRD_READ instrui *_bios_keybrd* a ler as teclas estendidas, tais como as teclas de seta do cursor.
_NKEYBRD_READY	Instrui *_bios_keybrd* a determinar se um caractere está presente no buffer do teclado. Se *_bios_keybrd* retorna 0, nenhuma tecla está presente. Se o valor de retorno é 0xFFFF, o usuário pressionou Ctrl+C. O valor _NKEYBRD_READY instrui *_bios_keybrd* a suportar as teclas estendidas, tais como as teclas de seta do cursor.
_NKEYBRD_SHIFTSTATUS	Instrui *_bios_keybrd* a retornar o estado de modificação do teclado, incluindo as teclas estendidas: Bit 15 Ligado — Tecla SysReq pressionada Bit 14 Ligado — Tecla CapsLock pressionada Bit 13 Ligado — Tecla NumLock pressionada Bit 12 Ligado — Tecla ScrollLock pressionada Bit 11 Ligado — Tecla Alt direita pressionada Bit 10 Ligado — Tecla Ctrl direita pressionada Bit 9 Ligado — Tecla Alt esquerda pressionada Bit 8 Ligado — Tecla Alt direita pressionada

O programa a seguir, *esttecla.c*, usa um laço para exibir as alterações no estado do teclado até que você pressione uma tecla diferente de Shift, Alt, Ctrl, NumLock e assim por diante. O programa somente lê as teclas não-estendidas, como mostrado aqui:

```
#include <stdio.h>
#include <bios.h>

void main(void)
 {
   unsigned int estado, antigo_estado = 0;

   do
     {
       estado = _bios_keybrd(_KEYBRD_SHIFTSTATUS);
       if (estado != antigo_estado)
         {
```

```
            antigo_estado = estado;
            if (estado & 0x80)
              printf("Ins ligada ");
            if (estado & 0x40)
              printf("Caps ligada ");
            if (estado & 0x20)
              printf("Num Lock ligada ");
            if (estado & 0x10)
              printf("Scroll Lock ligada ");
            if (estado & 0x08)
              printf("Alt pressionada ");
            if (estado & 0x04)
              printf("Ctrl pressionada ");
            if (estado & 0x02)
              printf("Shift esquerdo pressionada ");
            if (estado & 0x01)
              printf("Shift direito pressionada ");
            printf("\n");
          }
      }
      while (! _bios_keybrd(_KEYBRD_READY));
  }
```

Nota: Muitos compiladores C oferecem uma função chamada ***bioskey****, que efetua processamento similar a* ***_bios_keyboard****. Referencie a documentação que acompanha seu compilador para determinar qual função ele suporta.*

Nota: O Windows não suporta o comando ***_bios_keyboard*** *ou seus equivalentes. Dentro do Windows, você obterá informações sobre o teclado usando diferentes classes específicas do compilador. Por exemplo, dentro do ambiente do Borland C++ 5.02, você determina se a tecla Caps Lock e outras teclas não baseadas em digitação estão selecionadas usando a propriedade* ***KeyboardFlags*** *da classe* ***KeyboardManager****. Consulte a documentação do seu compilador para obter maiores informações sobre o controle de informações do teclado dentro do Windows.*

OBTENDO A LISTA DE EQUIPAMENTOS DA BIOS 563

À medida que seus programas aumentam em complexidade, algumas vezes eles precisam determinar detalhes específicos sobre o hardware do computador. Nesses casos, seus programas podem usar a função *_bios_equiplist*, como mostrado aqui:

```
#include <bios.h>

unsigned _bios_equiplist(void);
```

A função a seguir retorna um valor *unsigned int*, cujos bits têm o seguinte significado:

```
struct Equip
 {
  unsigned disquete_dispo:1;              // 1 se presente
  unsigned coprocessador_disponivel:1;    // 1 se presente
  unsigned memoria_sistema:2;             // Original do PC
  unsigned memoria_video:2;               // 01 = 40 x 25 mono
                                          // 10 = 80 x 25 cor
                                          // 11= 80 x 25 mono
  unsigned num_unidade_disquete:2;        // Soma 1
  unsigned num_portas_seriais:2;
  unsigned adaptador_jogos_disponivel:1; // 1 se presente
  unsigned num_impressoras:2;
 };
```

O programa a seguir, *exibequi.c*, usa a função *_bios_equiplist* para exibir a lista de equipamentos do sistema:

```
#include <stdio.h>
#include <bios.h>
```

```
void main(void)
 {
   struct Equip
    {
     unsigned disquete_dispo:1;
     unsigned coprocessador_disponivel:1;
     unsigned memoria_sistema:2;
     unsigned memoria_video:2;
     unsigned num_unidade_disquete:2;
     unsigned naousado_1:1;
     unsigned num_portas_seriais:3;
     unsigned adaptador_jogos_disponivel:1;
     unsigned naousado_2:1;
     unsigned num_impressoras:2;
    };
   union Equipamento
    {
       unsigned lista;
       struct Equip lista_bits;
    } equip;

   equip.lista = _bios_equiplista();
   if (equip.lista_bits.coprocessador_disponivel)
     printf("Co-processador matemático disponível\n");
   else
     printf("Sem co-processador matemático\n");
   printf("Memória na placa do sistema %d\n",
      (equip.lista_bits.memoria_sistema + 1) * 16);
   printf("Número de unidades de disquete %d\n",
      equip.lista_bits.num_unidade_disquete + 1);
   printf("Número de impressoras %d\n",
      equip.lista_bits.num_impressoras);
   printf("Número de portas seriais %d\n",
      equip.lista_bits.num_portas_seriais);
 }
```

Nota: Alguns compiladores C oferecem uma função chamada ***biosequip****, que efetua processamento similar a* ***_bios_equiplist****. O suporte para* ***_bios_equiplist*** *e comandos relacionados dentro do Windows varia de um compilador para outro. Por exemplo, o Borland C++ 5.02 suporta o comando* ***_bios_equiplist****, enquanto o Visual C++ usa as chamadas da API do Windows para obter informações do sistema. Veja os detalhes na documentação do seu compilador. O CD-ROM que acompanha este livro contém o programa* ***Win_Equi.cpp****, que usa a API do Windows para retornar informações do sistema.*

564 CONTROLANDO A E/S DA PORTA SERIAL

Para ajudar seus programas a efetuar operações de E/S em uma porta serial, tal como COM1, muitos compiladores baseados no DOS oferecem a função *bioscom*, como mostrado aqui:

```
#include <bios.h>

unsigned bioscom(int comando, int porta, char byte);
```

O parâmetro *comando* especifica a operação que você quer, e precisa ser um valor listado na Tabela 564.1

Tabela 564.1 As definições possíveis para o parâmetro ***comando***.

Valor	Descrição
_COM_INIT	Define as configurações de comunicação da porta
_COM_RECEIVE	Recebe um byte da porta
_COM_SEND	Envia um byte para a porta
_COM_STATUS	Retorna as configurações da porta

O parâmetro *porta* especifica a porta serial que você quer, onde 0 corresponde a COM1, 1 a COM2, e assim por diante. O parâmetro *byte* especifica um byte para saída ou as definições de comunicações que você quer. Se o valor contém as definições de comunicações que você quer, o parâmetro *byte* pode conter uma combinação dos valores listados na Tabela 564.2:

Tabela 564.2 Os valores possíveis para o parâmetro ***byte***.

Valor	Significado
_COM_CHR7	7 bits de dados
_COM_CHR8	8 bits de dados
_COM_STOP1	1 bit de parada
_COM_STOP2	2 bits de parada
_COM_NOPARITY	Sem paridade
_COM_ODDPARITY	Paridade ímpar
_COM_EVENPARITY	Paridade par
_COM_110	110 baud
_COM_150	150 baud
_COM_300	300 baud
_COM_600	600 baud
_COM_1200	1.200 baud
_COM_2400	2.400 baud
_COM_4800	4.800 baud
_COM_9600	9.600 baud

Independentemente do comando, o byte de alta ordem do valor de retorno tem os bits de significado listados na Tabela 564 3.

Tabela 564.3 Significado dos bits do valor de retorno para ***_bios_serialcom***.

Bit	Significado Se Ligado
8	Dados prontos
9	Erro de overrun
10	Erro de paridade
11	Erro de montagem do quadro
12	Break detectado
13	Registro transmit holding vazio
14	Registro transmit shift vazio
15	Tempo limite

Para _COM_INIT e _COM_STATUS, *_bios_serialcom* define o byte menos significativo do valor de retorno de acordo com os valores na Tabela 564.4.

Tabela 564.4 Os valores de retorno quando você usa ***COM_INIT*** e ***COM_STATUS***.

Bit	Significado Se Ligado
0	Mudança em clear to send
1	Mudança em data set ready
2	Detector de trailing-edge ring
3	Mudança no detector receive line signal
4	Clear to send
5	Data set ready
6	Indicador de chamada
7	Detecção received line signal

O programa a seguir, *setcom1.c*, define as comunicações de dados para a COM1 como 9.600 baud, 8 bits de dados, 1 bit de parada e sem paridade:

```
#include <stdio.h>
```

```
#include <bios.h>

void main(void)
 {
   char i = 0, titulo[]="Biblia do Programador C/C++";
   unsigned status;

   status = _bios_serialcom( COM_INIT, 0, _COM_9600 |
              _COM_CHR8 | _COM_STOP1 | _COM_NOPARITY);
   if (status & 0x100) // Dados prontos
     while (titulo[i])
       {
          _bios_serialcom(_COM_SEND, 0, titulo[i]);
          putchar(titulo[i]);
          i++;
       }
 }
```

Nota: Alguns compiladores C fornecem a função bioscom, que oferece processamento similar. O CD-ROM que acompanha este livro inclui o programa win_serial.cpp, que lê a porta serial a partir de dentro de programas Windows.

565 Acessando os Serviços do DOS Usando bdos

Como você aprendeu, a função *intdos* permite que seus programas acessem os serviços do DOS. Alguns serviços do DOS somente usam os registradores AX e DX. Para esses serviços, seus programas podem usar a função *bdos*, como mostrado aqui:

```
#include <dos.h>

int bdos(int dos_funcao, unsigned dx_reg, unsigned al_reg);
```

O parâmetro *dos_funcao* especifica o serviço que você quer chamar. Os parâmetros *dx_reg* e *al_reg* especificam os valores que o serviço espera nos registradores DX e AL. No retorno, a função retorna o valor do registrador AX ao término do serviço. O programa a seguir, *bdos.c*, usa a função *bdos* para exibir a unidade de disco atual:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   int drive;

   unid = bdos(0x19, 0, 0);
   printf("A unidade atual é %c\n", 'A' + unid);
 {
```

Nota: A função bdos passa um valor não-sinalizado para o registrador DX. Se você estiver usando um serviço do DOS que requer um ponteiro, poderá usar a função bdosptr. Se você usar o modelo de memória small, o segundo parâmetro corresponderá a DX. No modelo de memória large, o valor corresponderá a DS:DX.

566 Obtendo Informações Estendidas de Erro do DOS

Quando um serviço do sistema DOS falha, seus programas podem requerer informações adicionais para determinar a origem e a causa do erro. Para lhe ajudar a solicitar informações de erro estendidas, muitos compiladores C fornecem a função *dosexterr*, como mostrado aqui:

```
#include <dos.h>

int dosexterr(struct DOSERROR *erro_info);
```

O parâmetro *erro_info* é um ponteiro para uma estrutura do tipo *DOSERROR* que contém as informações estendidas de erro, como mostrado aqui:

```
struct DOSERROR
```

```
{
  int de_exterror;    // Erro estendido
  int de_class;       // Classe do erro
  int de_action;      // Ação recomendada
  int de_locus;       // Local do erro
```

Se a função *dosexterr* retornar 0, a chamada de serviço do DOS anterior não experimentou um erro. O valor estendido do erro fornece um erro específico. A classe do erro descreve a categoria do erro, como mostrado na Tabela 566.1.

Tabela 566.1 As classes de erro que ***dosexterr*** retorna dentro do membro ***de_class***.

Valor	Significado
01H	Sem recurso
02H	Erro temporário
03H	Erro de autorização
04H	Erro do sistema
05H	Falha do hardware
06H	Erro do sistema não devido ao programa atual
07H	Erro do aplicativo
08H	Item não-encontrado
09H	Formato inválido
0AH	Item bloqueado
0BH	Erro da *mídia*
0CH	O item já existe
0DH	Erro desconhecido

O membro *de_action* (a ação recomendada) diz ao programa como responder ao erro, como mostrado na Tabela 566.2.

Tabela 566.2 Os valores de retorno possíveis dentro do membro ***de_action***

Valor	Ação
01H	Repetir primeiro e depois avisar o usuário
02H	Repetir após retardo e depois avisar o usuário
03H	Pedir que o usuário dê uma solução
04H	Abortar com limpeza geral
05H	Abortar sem limpeza geral
06H	Ignorar o erro
07H	Repetir após a intervenção do usuário

Finalmente, o membro *de_locus* especifica a origem do erro, como mostrado na Tabela 566.3.

Tabela 566.3 Valores de retorno dentro do membro ***de_locus***.

Valor	Local
01H	Origem desconhecida
02H	Erro no dispositivo de bloco
03H	Erro de rede
04H	Erro de dispositivo serial
05H	Erro de memória

Quando seus programas precisam responder a erros de um modo cuidadoso, você deve usar a estrutura *dosexterr* para obter informações adicionais.

567 DETERMINANDO A QUANTIDADE DE MEMÓRIA CONVENCIONAL DA BIOS

Muitos programas mais antigos não fazem uso da memória estendida ou expandida. Em vez disso, os programas somente usam a memória convencional de 640Kb do PC. À medida que você examina esses programas, pode encontrar chamadas à função *biosmemory*, que retorna a quantidade de memória convencional (em Kb) que a BIOS informou durante a inicialização do sistema. A quantidade de memória que *biosmemory* retorna não inclui a memória estendida, expandida ou superior. Você implementará *biosmemory* como a seguir:

```
#include <bios.h>

int biosmemory(void);
```

Além da função *biosmemory*, você pode encontrar a função *_bios_memsize*, que efetua processamento idêntico. Você implementará *_bios_memsize* como mostrado aqui:

```
#include <bios.h>

int _bios_memsize(void);
```

O programa a seguir, *biosmem.c*, exibirá a quantidade de memória que a BIOS informa em resposta à chamada de *biosmemory* e *_bios_memsize*:

```
#include <stdio.h>
#include <bios.h>

void main(void)
 {
   printf("Relatório de memória da BIOS %dKb\n", biosmemory());
   printf("Relatório de memória da BIOS %dKb\n", _bios_memsize());
 }
```

Nota: Como o modelo de memória do Windows usa "memória virtual", você deve usar as chamadas da API do Windows detalhadas na seção Gerenciamento da Memória no Windows, mais à frente, ao desenvolver programas para o Windows.

568 CONSTRUINDO UM PONTEIRO FAR

Um *ponteiro far* consiste de um endereço de segmento de 16 bits e de um deslocamento de 16 bits. Quando você trabalha com ponteiros far, algumas vezes precisa dividir os ponteiros em suas partes de segmento e de deslocamento. Da mesma forma, algumas vezes você precisa criar um ponteiro far a partir de um endereço de segmento e de deslocamento. Para lhe ajudar a criar um ponteiro far, C oferece a macro *MK_FP*, como mostrado aqui:

```
#include <dos.h>

void far *MK_FP(unsigned segmento, unsigned deslocamento);
```

O seguinte fragmento de código usa *MK_FP* para criar um ponteiro far a partir de um endereço de variável near:

```
long far *fptr;
long variavel;
struct SREGS segs;

  //  Obtém o segmento de dados atual
segread(&segs);
fptr = MK_FP(segs.ds, &variavel);
```

Para compreender melhor a macro *MK_FP*, considere a seguinte implementação:

```
#define MK_FP(s,o) ((void far *) (((long) s << 16) | (o)))
```

Para criar o endereço far de 32 bits, a macro cria um valor *long* e desloca os bits do endereço do segmento para os 16 bits mais significativos do valor. Em seguida, a macro usa uma operação *OU* bit a bit para atribuir o endereço do deslocamento aos 16 bits menos significativos.

Nota: Como você aprendeu, como os ponteiros far não se aplicam ao modelo de memória virtual, os programas Windows não os utilizam.

DIVIDINDO UM ENDEREÇO FAR EM UM SEGMENTO E DESLOCAMENTO 569

Como discutido na dica anterior, um ponteiro far consiste de um segmento de 16 bits e um endereço de deslocamento de 16 bits. Quando você trabalha com ponteiros far, algumas vezes precisa dividir o endereço que o ponteiro far referencia em suas partes de segmento e de deslocamento. Nesses casos, seus programas podem usar as macros *FP_SEG* e *FP_OFF*, como mostrado aqui:

```
#include <dos.h>

unsigned FP_OFF(void far *ponteiro);
unsigned FP_SEG(void far *ponteiro);
```

Os comandos a seguir ilustram o uso das macros *FP_SEG* e *FP_OFF*:

```
char far *titulo = "Bíblia do Programador C/C++";
unsigned segmento, desloc;

segmento = FP_SEG(titulo);
desloc = FP_OFF(titulo);
```

Nota: Como você aprendeu, como os ponteiros far não se aplicam ao modelo de memória virtual, os programas Windows não os utilizam.

DETERMINANDO A MEMÓRIA DO NÚCLEO LIVRE 570

À medida que seus programas alocam memória, você pode usar a função *coreleft* para estimar a quantidade de memória convencional atualmente disponível para alocação. A função *coreleft* não fornece um informe exato da memória não-usada. Em vez disso, se você estiver usando o modelo de memória small, *coreleft* retornará a memória não-usada entre o topo do heap e a pilha. Se você estiver usando um modelo de memória maior, *coreleft* retornará a quantidade de memória entre o topo da memória alocada e o final da memória convencional. A função *coreleft* retornará a memória não-usada em bytes. No caso do modelo de memória small, *coreleft* retornará um valor sem sinal, como a seguir:

```
#include <alloc.h>

unsigned coreleft(void);
```

Se você estiver usando um modelo de memória maior, *coreleft* retornará um valor do tipo *long*, como mostrado aqui:

```
#include <alloc.h>

long coreleft(void);
```

O programa a seguir, *coreleft.c*, exibe a quantidade de memória disponível. O programa usa as constantes do modelo de memória que muitos compiladores suportam para determinar o modelo de memória atual:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
   #if defined(__SMALL__)
```

```
    unsigned result;
  #else
    long result;
  #endif
  result = coreleft();
  printf("A quantidade de memória disponível é %dKb\n", result / 1024);
}
```

Nota: Se seu compilador não suporta a função coreleft, verifique se ele fornece as funções _memavl e _memmax. A seção Serviços DOS e BIOS, mais à frente, detalhará como determinar a memória disponível dentro do ambiente Windows.

571 LENDO AS CONFIGURAÇÕES DO REGISTRADOR DE SEGMENTO

Quando você trabalha dentro do ambiente do DOS, o compilador controla o código do seu programa, os dados e a pilha usando quatro registradores de segmento. Os quatro registradores de segmento são listados na Tabela 571.

Tabela 571 Os quatro registradores de segmento do PC

Nome	*Descrição*
cs	Registrador segmento de código
ds	Registrador segmento de dados
ss	Registrador segmento de pilha
es	Registrador segmento extra

Dependendo do modelo de memória do seu programa, cada registrador de segmento pode apontar para um segmento exclusivo de 64Kb, ou dois ou mais registradores de segmento podem apontar para o mesmo segmento. Quando seus programas usam os serviços do DOS e da BIOS, algumas vezes você precisa conhecer o valor de um registrador de segmento. Para esses casos, você pode usar a função *segread*, como mostrado aqui:

```
#include <dos.h>

void segread(struct SREGS *segs);
```

O arquivo de cabeçalho *dos.h* define a estrutura *SREGS*, como mostrado aqui:

```
struct SREGS
  {
   unsigned int es;
   unsigned int cs;
   unsigned int ss;
   unsigned int ds;
  };
```

O programa a seguir, *exibeseg.c*, usa a função *segread* para exibir o conteúdo atual dos registradores de segmento:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct SREGS segs;

   segread(&segs);
   printf("CS %X DS %X SS %X ES %X\n", segs.cs, segs.ds, segs.ss, segs.es);
 }
```

COMPREENDENDO OS TIPOS DE MEMÓRIA 572

O PC pode ter três tipos de memória: convencional, estendida e expandida. Várias dicas a seguir discutem esses tipos de memória em detalhe. À medida que você programa, é importante que compreenda os diferentes tipos de memória e suas características. Os passos que você precisa executar para alocar e usar os diferentes tipos de memória serão diferentes. Além disso, cada tipo de memória tem uma velocidade de acesso diferente, o que afetará o desempenho do seu programa. Para determinar a quantidade e os tipos de memória instalados no seu PC, você pode usar o comando MEM /CLASSIFY do DOS 5 (ou posterior), como mostrado aqui:

```
C:\> MEM /CLASSIFY <Enter>
```

Se você não estiver usando o DOS 5 ou posterior, deverá atualizar seu sistema. O DOS 5 oferece várias capacidades de gerenciamento de memória que lhe ajudam a maximizar o uso de memória do seu PC.

COMPREENDENDO A MEMÓRIA CONVENCIONAL 573

Quando a IBM lançou o PC em 1981, o computador normalmente vinha com 64Kb e 256Kb de RAM. Naquele tempo, essa memória era mais do que suficiente. Essa memória ficou conhecida como a *memória convencional* do PC. Hoje, a memória convencional do PC é o primeiro 1Mb de RAM. Os programas do DOS tipicamente rodam dentro dos primeiros 640Kb de memória convencional. O PC usa os 384Kb de memória (chamada *memoria reservada* ou *memória superior*) que res.de entre 640Kb e 1Mb para a memória de vídeo do computador, os controladores de dispositivo, outros dispositivos de hardware mapeados na memória e a BIOS. No entanto, durante anos, o sistema operacional não usou grandes seções dessa memória reservada. Começando com a versão 5, o DOS fornece modos para que seus programas e controladores de dispositivos possam residir nas regiões não usadas enquanto seus programas são executados. Utilizar a memória superior lhe permite liberar mais da memória convenciona. de 640Kb para o DOS usar. Para informações sobre como fazer uso da região de memória superior, consulte a descrição do comando DOS=UMB (upper memory block — *bloco de memória superior*) do CONFIG.SYS na documentação do DOS.

Como já foi visto, o Windows usa o modelo de memória virtual para gerenciar a memória, o que significa que as questões de memória convencional não são significativas nos programas Windows. No entanto, a memória convencional é importante quando você roda programas dentro de uma janela do DOS sob o Windows.

COMPREENDENDO O LAYOUT DA MEMÓRIA CONVENCIONAL 574

Na dica anterior você aprendeu que memória convencional é o primeiro 1Mb de RAM do seu computador. Seus programas e o DOS normalmente residem nos primeiros 640Kb de memória convencional. Para lhe ajudar a compreender melhor como o DOS usa a memória convencional, a Figura 574 apresenta um mapa da memória convencional.

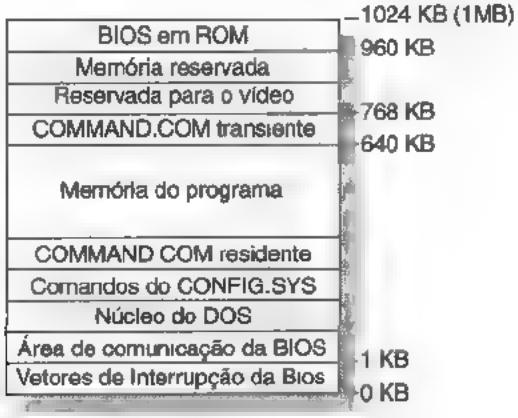

Figura 574 Mapa da memória convencional do PC.

A seção Serviços DOS e BIOS, anteriormente explicou os vetores de interrupção da BIOS e a área de comunicação da BIOS. O *núcleo* do DOS é o software, *io.sys* e *msdos.sys*, que o DOS carrega na memória durante

a inicialização do sistema. Os comandos do CONFIG.SYS representam a região de memória que o DOS aloca para os controladores de dispositivo, buffers do disco, e assim por diante. As áreas *command.com residente* e *command.com transiente* contêm o software responsável por exibir o prompt do DOS, e processar os comandos que você digita. Para tornar mais memória, disponível aos programas, o DOS divide o *command.com* em uma seção residente, que sempre permanece na memória e uma seção transiente, que cada comando pode sobrescrever. Após o comando terminar, a parte residente do *command.com* recarrega a seção transiente do disco. Os 384Kb de memória entre os 640Kb e 1Mb é a memória, superior do seu computador, que contém a memória de vídeo, os blocos de memória superior, e os serviços da BIOS baseados em ROM, como já discutido na seção DOS e BIOS.

Nota: As considerações de memória convencional não são tão importantes dentro do Windows 95 e do Windows NT como são dentro do DOS. Como o Windows usa o modelo de memória virtual, a maior parte da execução do seu programa ocorrerá dentro do primeiro 1Mb de RAM — ou nas localizações superiores da RAM, ou na memória virtual que o computador usa no disco rígido. Dicas posteriores discutirão em detalhes o modelo de memória virtual.

575 Acessando a Memória Convencional

Em resumo, o *modelo de memória* do seu programa define o uso de memória convencional do programa. Dependendo do modelo de memória que seu programa usar, o compilador alocará um ou mais segmentos de 64Kb para armazenar o código e os dados do programa. Quando seu programa precisar alocar memória dinamicamente, ele poderá usar as funções de C, tais como *malloc*, para alocar memória no heap near, ou *_fmalloc*, para alocar memória no heap *far*. As Dicas 597 e 598 discutem os heaps do tipo near e far. Além disso, seus programas podem usar os serviços do sistema DOS para alocar memória.

Nota: Como uma regra, seus programas devem usar somente um método para alocar e desalocar memória. Para melhorar a portabilidade, seus programas devem usar as funções da biblioteca de execução de C para o gerenciamento da memória. Não misture as funções de alocação de memória de C com aquelas que o DOS oferece. Combinando as funções de alocação de memória de C e do DOS, você aumenta a possibilidade de erros, e torna seus programas mais difíceis de compreender.

576 Compreendendo Por Que o PC e o DOS Estão Restritos a 1Mb

Muitas pessoas freqüentemente referenciam a *barreira de 640Kb* quando discutem o DOS. Em resumo, a barreira de 640Kb referencia a região de memória convencional dentro da qual seus programas precisam rodar. No entanto, como você aprendeu, os programas do DOS na verdade usam os serviços da BIOS e a memória de vídeo, que residem no intervalo de memória de 640Kb a 1Mb. Além disso, começando com o DOS 5, seus programas e controladores de dispositivo podem realmente residir na área de memória superior, de modo que a restrição de memória do DOS na verdade ocorre em 1Mb.

O limite de memória de 1Mb é mais um limite do PC do que do DOS. O PC original (que usava o processador 8088) usava um endereço de segmento de 16 bits e um deslocamento de 16 bits dentro do segmento. Dentro da memória do PC, os segmentos ocorrem em intervalos de 16 bytes. Os 65.535 endereços de segmento distintos permitem que o PC acesse 65.536 * 16 bytes (1.048.576) posições de memória diferentes. Como o DOS precisa rodar dentro desse ambiente, ele injustamente leva a culpa por restringir a memória do programa.

Nota: O Windows usa um tipo de 32 bits especial conhecido como DWORD para armazenar os endereços de segmento e de deslocamento. A DWORD de 32 bits permite que o Windows acesse até 4Gb de RAM, desde que o processador do computador possa acessar isso tudo. A maioria dos computadores equipados com o processador Pentium pode acessar até 128Mb de RAM.

577 Produzindo um Endereço a partir de Segmentos e Deslocamentos

Para gerenciar as posições de memória, o PC usa o endereço de *segmento* e de *deslocamento* de 16 bits. O endereço do segmento normalmente identifica o início de uma região de 64Kb. O endereço de deslocamento identifica um byte específico dentro da região. Os segmentos podem começar em intervalos de 16 bytes chamados *parágrafos*. Para endereçar a memória, o PC combina o endereço de segmento e de deslocamento para produzir um

endereço de 20 bits, que pode endereçar 1.048.576 posições diferentes de memória (1Mb). Para criar o endereço de 20 bits, o PC desloca o endereço do segmento de 16 bits quatro bits para a esquerda, e, depois, soma ao resultado o endereço do deslocamento. Por exemplo, assuma que o endereço do segmento seja 1234H. Quando o PC desloca o endereço para a esquerda, o resultado torna-se 12340H. Em seguida, se o endereço do deslocamento for 5, o resultado será 12340H + 5 ou 12345H. A equação a seguir ilustra melhor o processamento envolvido:

```
1234H   O segmento deslocado torna-se 12340H
Somando o deslocamento de     0005H
Resulta em 12345H
```

Se você examinar a operação em binario, o resultado torna-se o seguinte:

```
     0001 0010 0011 0100       Segmento
0001 0010 0011 0100 0000       Deslocado
                    0101       Deslocamento
=======================
0001 0010 0011 0100 0101       Resultado (endereço de 20 bits)
```

COMPREENDENDO A MEMÓRIA EXPANDIDA 578

Como você aprendeu, os programas do DOS normalmente rodam dentro da memória convencional de 640Kb do seu computador. No entanto, muitos programas maiores, tais como planilhas, requerem mais do que 640Kb. O IBM-PC original (8088) não podia endereçar memória além de 1Mb. Para permitir que o PC acessasse mais do que 1Mb de memória, as companhias Lotus, Intel e Microsoft criaram uma *especificação de memória expandida* (EMS), que combina software e uma placa de expansão de memória especial para fazer o PC acessar grandes quantidades de memória.

Para usar a memória expandida, seu computador precisa ter uma placa de memória expandida. Para começar, o software de especificação de memória expandida aloca um bloco de 64Kb dentro da memória superior (a região de 384Kb entre 640Kb e 1Mb). Em seguida, o software divide a região de 64Kb em quatro seções de 16Kb chamadas *páginas*. Quando seu programa inicia, ele usa funções especiais da especificação de memória expandida para alocar e carregar a memória expandida. Para fazer isso, seu programa define páginas lógicas (16Kb) dentro da região de memória expandida.

Por exemplo, se você tem uma planilha de 128Kb, o computador divide os dados em oito páginas lógicas de 16Kb. Quando seu programa precisa acessar uma página lógica específica, ele usa uma função da especificação de memória expandida para mapear a página lógica em uma das páginas da especificação de memória expandida na memória superior do seu computador, que o seu programa do DOS pode então acessar diretamente. À medida que seu programa usa outras páginas lógicas, ele mapeia as páginas para dentro e para fora da área da especificação de memória expandida, conforme for necessário.

Muitos programas do DOS requerem mapeamento da memória expandida somente porque o processador 8088 não poderá acessar as posições de memória além de 1Mb. Embora a memória expandida ofereça um modo para o 8088 acessar grandes quantidades de dados, o mapeamento contínuo dos dados introduz uma sobrecarga considerável, que degrada o desempenho do sistema. Se você estiver usando um 80286 ou superior, seu computador poderá acessar memória além de 1Mb (chamada memória estendida), o que é um processo muito mais rápido.

Nota: *Quando um programa usa memória expandida, o código do programa permanece na região de memória convencional de 640Kb. Somente os dados podem residir na área de memória expandida.*

Nota: *Como você aprendeu na Dica 574, as considerações de memória convencional não são tão importantes dentro do Windows 95 e do Windows NT como são dentro do DOS. Como a memória convencional não é crucial, a memória expandida também não é tão crucial para os computadores mais novos, seja rodando o Windows ou o DOS. Na verdade, os processadores mais recentes (x486 ou superiores) não suportam mais a memória expandida, somente a memória estendida, como será detalhado na Dica 580.*

USANDO A MEMÓRIA EXPANDIDA 579

A dica anterior introduziu o uso de memória expandida do PC. Como regra, para aumentar o desempenho, seus programas devem usar *memória estendida* (veja a dica a seguir). No entanto, se as circunstâncias lhe forçarem a

desenvolver um programa que precise rodar em um PC mais antigo, baseado no 8088, seus programas acessarão os serviços da especificação de memória expandida usando a função *int86* e a INT 67H, como detalhou a seção Serviços DOS e BIOS. Existem muitos serviços diferentes da especificação de memória expandida que lhe permitem alocar, mapear, desalocar e manipular a memória expandida. Para uma descrição completa e programas de exemplo que usam esses serviços, consulte o livro DOS Programming: The Complete Reference, *de Kris Jamsa, Osborne/McGraw Hill, 1991.* No entanto, o CD-ROM que acompanha este livro oferece um programa de exemplo, *showems.c*, que ilustra como você pode usar a memória expandida dentro de seus programas. A função a seguir, *checa_ems*, usa registradores e operações bit a bit para determinar se seu programa carregou os controladores de memória expandida:

```
int checa_ems(void)
 {
   union REGS ent_regs, sai_regs;
   struct SREGS segs;
   int grande, pequeno;   // Versão do DOS
   struct DeviceHeader   {
      struct DeviceHeader far *link;
      unsigned atributos;
      unsigned estrategia_desloc;
      unsigned interrup_desloc;
      char nome_ou_num_de_unidades[8];
   } far *dev;
   int i;
   char nome_contr[9];

   // Lê a versão do DOS
   ent_regs.x.ax = 0x3001;
   intdos (&ent_regs, &sai_regs);
   grande = sai_regs.h.al;
   pequeno = sai_regs.h.ah;
   if (grande < 2)
       return(0);      // Requer o DOS 2.0
   else
     {
   // Lê a lista das listas
   ent_regs.h.ah = 0x52;
   intdosx (&ent_regs, &sai_regs, &segs);
   if (grande == 2)
           dev = (struct DeviceHeader far *)
           MK_FP(segs.es + 1, sai_regs.x.bx + 7);
       else if ((grande == 3) && (pequeno == 0))
          dev = (struct DeviceHeader far *)
          MK_FP(segs.es + 2, sai_regs.x.bx + 8);
       else
          dev = (struct DeviceHeader far *)
           MK_FP(segs.es + 2, sai_regs.x.bx + 2);
       while (FP_OFF(dev) != 0xFFFF)
         {
            if (dev->atributos & 0x8000)
            {  // Dispositivo de caractere
              for (i = 0; i < 8; i++)
              nome_contr[i] = dev->nome_ou_num_de_unidades[i];
              nome_contr[8] = NULL;
            }
            if (! strcmp(nome_contr, "EMMXXXX0"))
           return(1);   // Controlador encontrado
         dev = dev->link;
      }
        }
   return(0);
 }
```

COMPREENDENDO A MEMÓRIA ESTENDIDA 580

O IBM-PC original (8088) usava endereçamento de 20 bits, o que restringia o acesso a 1Mb de memória. Começando com o IBM PC-AT (80286), o PC ganhou a capacidade de usar endereçamento de 24 bits, o que o permitiu endereçar até 16Mb. As máquinas baseadas no 386, 486, 586 e 686 aumentaram o endereçamento para 32 bits, o que permite que o PC enderece até 4Gb de memória. Quando o PC ganhou a capacidade de acessar memória além de 1Mb, os programadores chamaram a memória além de 1Mb de *memória estendida*. Como o IBM-PC original não pode acessar memória além de 1Mb, ele não pode usar a memória estendida.

Para acessar a memória estendida, você precisa carregar um controlador de dispositivo de memória estendida. No DOS, o controlador geralmente é *himem.sys*. Quando seus programas baseados no DOS usam memória estendida, somente os dados do programa podem residir na memória estendida. O código do programa precisa residir na memória convencional de 640Kb. No entanto, quando seus programas usam memória estendida, os serviços do sistema que fornecem acesso à memória precisam alterar o modo de execução da sua CPU do modo real para o modo protegido e de volta novamente. Alterar os modos da CPU requer um certo tempo de processamento, o que introduz sobrecarga. No entanto, a sobrecarga é menos que aquela da memória expandida — tornando a memória estendida mais desejável.

Nota: Como você aprendeu na Dica 574, as considerações de memória convencional não são tão importantes dentro do Windows 95 e do Windows NT como são dentro do DOS. Os computadores que rodam o Windows 95 ainda usam a assim chamada "memória estendida" para gerenciar o acesso à memória além do primeiro 1Mb do computador. Os computadores que rodam o Windows NT gerenciam a memória sem o uso transparente de memória estendida que o Windows 95 requer. Como você aprenderá em dicas posteriores, seus programas devem usar o modelo de memória virtual para gerenciar toda a memória a partir de dentro dos programas Windows.

COMPREENDENDO OS MODOS REAL E PROTEGIDO 581

O DOS é um sistema operacional monotarefa, o que significa (com a exceção dos controladores de dispositivo e os programas residentes na memória) que normalmente apenas um programa está em execução em um determinado momento. Como o DOS é um sistema operacional monotarefa, proteger um programa do outro não é uma questão importante. Portanto, o DOS permite que os programas acessem a memória do modo como desejarem. Em outras palavras, um programa baseado no DOS pode modificar o valor em qualquer posição na memória convencional. Quando você roda múltiplos programas ao mesmo tempo, um programa não pode aleatoriamente alterar a memória como pode dentro de um ambiente monotarefa, pois o programa provavelmente sobrescreveria o conteúdo de outro programa na memória. Em um ambiente de múltiplos programas, o sistema operacional precisa proteger a memória de um programa das ações do outro programa. Para proteger os programas na memória, o sistema operacional utiliza uma proteção de memória baseada no hardware.

Começando com o processador 80286, a CPU pode operar em dois modos: *real* e *protegido*. O modo real existe para compatibilidade com o IBM-PC original baseado no 8088. O DOS usa o modo real, que não tem proteção de memória. Outros sistemas operacionais, tais como o UNIX, OS/2, ou Windows, podem rodar no modo protegido. No modo protegido, um programa não pode acessar a memória de outro programa. Além disso, dentro do modo protegido, o PC muda de seu esquema de endereçamento segmento/deslocamento para outro em que a CPU usa endereçamento de 24 bits dentro do 80286 e 32 bits nas máquinas equipadas com o 80386 (e superiores). Assim, o modo permite que o PC enderece memória estendida para código e dados.

Quando seus programas baseados no DOS usam memória estendida, o software que eles usam para acessar a memória estendida alterna transparentemente a CPU do modo real (que estava rodando o DOS) para o modo protegido (que pode acessar a memória estendida) e depois volta para o modo real outra vez.

ACESSANDO A MEMÓRIA ESTENDIDA 582

Antes que seus programas possam usar a memória estendida, você precisará instalar um controlador de dispositivo de memória estendida (normalmente *himem.sys*). Em seguida, usando a interrupção multiplex do DOS, INT 2FH, serviço 4300H, seus programas podem obter um ponto de entrada na memória do computador para os serviços de memória estendida. O controlador de memória estendida fornece funções que permite que seus

programas aloquem, desaloquem e manipulem a memória estendida. Para acessar os serviços, você atribui diferentes parâmetros aos registradores do PC, e, depois, salta para o ponto de entrada especificado. Para uma descrição completa dos serviços de memória estendida, veja o livro *DOS Programming: The Complete Reference*, de Kris Jamsa, Osborne/McGraw Hill, 1991. Para lhe ajudar a compreender melhor os serviços de memória estendida, o CD-ROM que acompanha este livro inclui o arquivo *xmsdemo.c*.

583 COMPREENDENDO A ÁREA DE MEMÓRIA ALTA

Como você aprendeu, a memória estendida é a memória do seu computador acima de 1Mb. Quando os programas do DOS acessam a memória estendida, a CPU muda do modo real para o modo protegido e de volta outra vez. Se você estiver usando um computador 386 e o DOS 5 ou posterior, poderá utilizar um "defeito" no projeto do processador 386 que lhe permitirá acessar os primeiros 64Kb de memória estendida a partir de dentro do modo real. Como mostrado na Figura 583, essa região de 64Kb é chamada *área de memória alta*.

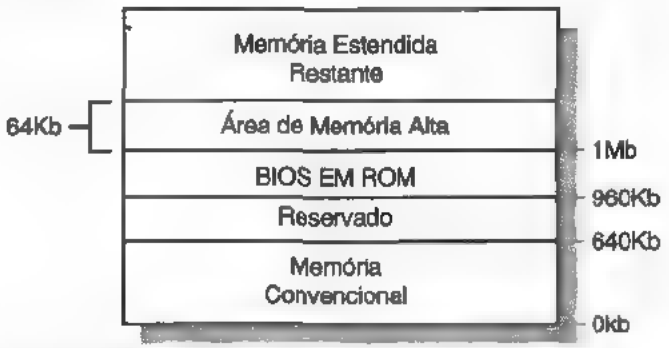

Figura 583 A área de memória alta são os primeiros 64Kb da memória estendida.

O melhor modo de usar a área de memória alta é carregar o núcleo do DOS nela, liberando a memória dentro da memória convencional de 640Kb. No entanto, se o DOS não estiver usando a área de memória alta, seu programa poderá alocá-la usando um serviço de memória estendida. Para carregar o DOS na área de memória alta, você precisa instalar o controlador *himem.sys*, e, depois, usar o comando DOS=HIGH no arquivo *config.sys*

Nota: Como esta dica indica, seus programas não deverão usar a Área de Memória Alta se eles forem rodar em computadores mais recentes que o x386. Se rodam sob o Windows 95 ou Windows NT, seus programas também não devem usar a Área de Memória Alta.

584 COMPREENDENDO A PILHA

A *pilha* é uma região de memória dentro da qual seus programas temporariamente armazenam dados quando são executados. Por exemplo, quando seus programas passam parâmetros para as funções, C coloca os parâmetros na pilha. Quando a função termina, C remove os itens da pilha. Da mesma forma, quando suas funções declaram variáveis locais, C armazena os valores das variáveis na pilha durante a execução da função. Quando a função termina, C descarta as variáveis.

A pilha tem esse nome porque os programas *colocam* valores na pilha, de forma muito parecida com bandejas de café empilhadas uma sobre a outra em uma lanchonete. Dependendo do modelo de memória do programa, a quantidade de espaço na pilha que o compilador fornece irá variar. Dependendo do uso de funções e parâmetros de seus programas, a quantidade de espaço na pilha que seu programa requer irá variar. No mínimo, o compilador alocará 4Kb de espaço na pilha Se seu programa precisar de mais ou menos espaço, você poderá usar diretivas do compilador e do linkeditor para controlar o espaço que será alocado na pilha. O PC usa dois registradores para localizar a pilha. O registrador segmento de pilha (SS) aponta para o início da pilha e o registrador Ponteiro da Pilha (SP) aponta para o topo da pilha.

COMPREENDENDO DIFERENTES CONFIGURAÇÕES DA PILHA 585

Na dica anterior vimos que seu programa usa a pilha para armazenar informações de forma temporária — principalmente durante as chamadas das funções. Dependendo do uso de funções no seu programa e do número e tamanho dos parâmetros que o programa passar para essas funções, a quantidade de espaço na pilha que seu programa requer irá variar de um programa para outro. Quando você usar o modelo de memória small, C alocará espaço no topo do segmento de dados, como mostrado na Figura 585.

Figura 585 Alocação de espaço na pilha no modelo de memória small.

Por outro lado, quando você usar o modelo de memória large ou compact, C alocará um segmento inteiro de 64Kb para a pilha. Caso seu programa coloque mais informações na pilha que esta poderá conter, um *erro de extravasamento de pilha* ocorrerá. Se seu programa desabilitou a verificação da pilha, você não saberá do erro, e os dados que colocou na pilha poderão sobrescrever os dados do programa. A dica a seguir apresenta modos de você determinar o tamanho da pilha atual do seu programa.

Nota: Você aprenderá mais sobre a verificação da pilha na seção Otimização.

O Windows, no entanto, constrói a pilha de forma um pouco diferente de uma pilha do DOS. A pilha do Windows, por padrão, é de 1Mb, e seu limite de tamanho é o limite da memória virtual; o que significa que uma pilha poderia ter até 250Mb ou mais. Portanto, o tamanho da pilha de 250Mb reduz sua preocupação em proteger os programas de erros na pilha.

DETERMINANDO O TAMANHO DA PILHA ATUAL DO SEU PROGRAMA 586

Dependendo do uso de funções e de parâmetros do seu programa, a quantidade de espaço na pilha que seu programa requer irá variar. Usando as diretivas do compilador e do linkeditor, seus programas podem alocar um tamanho de pilha específico. À medida que seus programas são executados, algumas vezes você desejará saber o tamanho atual da pilha. Se você estiver usando o Turbo C++ Lite, poderá usar a variável global *_stklen*. O programa a seguir, *tampilha.c*, usa a variável global *_stklen* para exibir o tamanho atual da pilha:

```
#include <stdio.h>
#include <dos.h>

void main(void)
  {
   printf("O tamanho atual da pilha é %d bytes\n", _stklen);
  }
```

Se você estiver usando o Microsoft Visual C++, a função *stackavail* retornará a quantidade de espaço disponível na pilha.

CONTROLANDO O ESPAÇO NA PILHA COM _STKLEN 587

Na dica anterior você aprendeu que a variável global *_stklen* permite que seus programas determinem o tamanho atual da pilha. Além disso, seus programas podem usar a variável *_stklen* para controlar a quantidade de espaço na pilha que o compilador aloca. Para especificar um tamanho de pilha usando *_stklen*, seus programas precisam declarar a variável como uma variável global externa. O programa a seguir, *pilha8k.c*, usa a variável global *_stklen* para alocar uma pilha de 8Kb:

```
#include <stdio.h>
```

```
#include <dos.h>

extern unsigned _stklen = 8096;

void main(void)
  {
   printf("Tamanho atual da pilha é %d bytes\n", _stklen);
  }
```

Nota: Dentro do Windows, a maioria dos compiladores permite que você defina o tamanho da pilha que um certo encadeamento requer dentro do comando que cria o encadeamento. A seção Gerenciamento de Memórias discutirá os encadeamentos em detalhe.

588 ATRIBUINDO UM VALOR A UM INTERVALO DE MEMÓRIA

Quando seus programas trabalham com matrizes e ponteiros para intervalos de memória, algumas vezes você deseja inicializar a memória com um determinado valor. Para fazer isso, seus programas podem usar a função *memset*. Você implementará a função *memset* como mostrado aqui:

```
#include <mem.h>

void *memset(void *ptr, int caractere, size_t num_bytes);
```

O parâmetro *ptr* é um ponteiro para o primeiro byte no intervalo de memória. O parâmetro *caractere* é o valor byte que você quer atribuir ao intervalo de memória. Finalmente, o parâmetro *num_bytes* especifica o número de bytes no intervalo de memória. A função retorna um ponteiro para o início do intervalo de memória. O comando a seguir usa a função *memset* para inicializar uma matriz de string de caracteres como *NULL*:

```
char string[128];

memset (string, NULL, sizeof(string));
```

589 COPIANDO UM INTERVALO DE MEMÓRIA PARA OUTRO

Quando seus programas trabalham com strings de caracteres, eles podem usar a função *strcpy* para copiar o conteúdo de uma string para outra. No entanto, quando você precisa copiar uma matriz de valores inteiros ou ponto flutuante, seus programas podem efetuar processamento similar usando as funções *memmove* ou *memcpy*, como mostrado aqui:

```
#include <mem.h>

void *memmove(void *alvo, const void *origem, size_t num_bytes);
void *memcpy(void *alvo, const void *origem, size_t num_bytes);
```

Os parâmetros *alvo* e *origem* são ponteiros para a matriz na qual a função copia os dados (*alvo*) e a partir da qual a função cria a cópia (*origem*). O parâmetro *num_bytes* especifica o número de bytes a copiar. A principal diferença entre as duas funções é que *memmove* corretamente copia os dados entre dois intervalos de bytes que podem se sobrepor na memória, enquanto *memcpy* pode copiar os dados incorretamente. O programa a seguir, *memmove.c*, usa a função *memmove* para copiar o conteúdo de uma matriz em ponto flutuante:

```
#include <stdio.h>
#include <mem.h>

void main(void)
  {
   float valores[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
   float vazio[5];
   int i;

   memmove(vazio, valores, sizeof(valores));
   for (i = 0; i < 5; i++)
     printf("%3.1f ", vazio[i]);
  }
```

COPIANDO UM INTERVALO DE MEMÓRIA ATÉ UM BYTE ESPECÍFICO 590

Quando seus programas trabalham com matrizes, algumas vezes você precisa copiar o conteúdo de uma matriz para outra. Dependendo do conteúdo da matriz, você poderá querer copiar para mover até *n* bytes, ou terminar imediatamente se a cópia encontrar um determinado caractere. Para efetuar tal processamento, seus programas podem usar a função *memccpy*, como mostrado aqui:

```
#include <mem.h>

void *memccpy(void *alvo, const void *origem, int caractere, size_t num_
bytes);
```

Os parâmetros *alvo* e *origem* são ponteiros para a matriz na qual a função está copiando os dados (*alvo*) e a partir do qual a função está fazendo a cópia (*origem*). O parâmetro *caractere* especifica o caractere que, se copiado, imediatamente finaliza a operação de cópia. O parâmetro *num_bytes* especifica o número de bytes a copiar. Se a função copiar *num_bytes*, ela retornará o valor *NULL*. Se a função encontrar o caractere que você especificar, ela retornará um ponteiro para o byte no alvo que imediatamente segue o caractere. O programa a seguir, *memccpy.c* usa a função *memccpy* para copiar as letras de A até K para a matriz *alvo*:

```
#include <stdio.h>
#include <mem.h>

void main(void)
  {
   char alfabeto[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   char alvo[27];
   char *result;

   result = memccpy(alvo, alfabeto,'K', sizeof(alfabeto));
   if (result)
      *result = NULL;
   printf(alvo);
  }
```

COMPARANDO DUAS MATRIZES DE UNSIGNED CHAR 591

Quando seus programas trabalham com matrizes, algumas vezes você irá querer compara dois intervalos de memória. Você mais comumente comparará dois intervalos de memória para verificar duas strings de caracteres. Para comparar dois intervalos de memória, seus programas podem usar as funções *memcmp* ou *memicmp*, como mostradas aqui:

```
#include <mem.h>

int memcmp(const void *bloco_1, const void *bloco_2, size_t num_bytes);
int memicmp(const void *bloco_1, const void *bloco_2, size_t num_bytes);
```

A diferença entre as funções *memcmp* e *memicmp* é que *memicmp* ignora a caixa das letras. Os parâmetros *bloco_1* e *bloco_2* são ponteiros para o início de cada intervalo de memória. O parâmetro *num_bytes* especifica o número de bytes a comparar. A função retorna um dos valores listados na Tabela 591.

Tabela 591 Os valores de retorno ***memcmp*** e ***memicmp***

Valor	Significado
Menor que 0	*bloco_1* é menor que *bloco_2*
0	os blocos são iguais
Maior que 0	*bloco_1* é maior que *bloco_2*

O programa a seguir, *memcmp.c*, usa as funções *memcmp* e *memicmp* para comparar duas strings de caracteres:

```
#include <stdio.h>
```

```
#include <mem.h>

void main(void)
  {
   char *a = "AAA";
   char *b = "BBB";
   char *c = "aaa";

   printf("Comparando %s e %s com memcmp %d\n",
          a, b, memcmp(a, b, sizeof(a)));
   printf("Comparando %s e %s com memicmp %d\n",
          a, b, memicmp(a, c, sizeof(a)));
 }
```

592 PERMUTANDO BYTES DE STRINGS DE CARACTERES ADJACENTES

Quando você trabalha com diferentes tipos de computadores, algumas vezes precisa permutar bytes adjacentes de memória. Para fazer isso, seus programas podem usar a função *swab*, como mostrado aqui:

```
#include <stdlib.h>

void swab(char *origem, char *alvo, int num_bytes);
```

O parâmetro *origem* é um ponteiro para uma string cujos bytes você quer permutar. O parâmetro *alvo* é um ponteiro para uma string para a qual *swab* atribui os bytes que permutou. O parâmetro *num_bytes* especifica o número de bytes a permutar. O programa a seguir, *swab.c*, ilustra a função *swab*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mem.h>

void main(void)
  {
   char *origem = "Bíblia do Programador C/C++";
   char alvo[64];

   memset(alvo, NULL, sizeof(alvo));
   swab(origem, alvo, strlen(origem));
   printf("Origem: %s Alvo %s\n", origem, alvo);
  }
```

593 ALOCANDO MEMÓRIA DINÂMICA

Quando seus programas declaram uma matriz, o compilador C aloca memória para armazenar a matriz. Se os requisitos do programa mudarem e o tamanho da matriz precisar aumentar ou diminuir, você precisará editar e recompilar o programa. Para reduzir o número de alterações que você precisa fazer nos seus programas para alterações nos tamanhos das matrizes, seus programas podem alocar sua própria memória durante a execução. Quando você aloca memória dessa forma, a biblioteca de execução de C retorna um ponteiro ao início do intervalo de memória. Seus programas podem então trabalhar com memória usando um formato de matriz ou de ponteiro, o que você preferir. Quando você aloca memória durante a execução, seus programas podem usar a função de biblioteca *malloc*, como mostrado aqui:

```
#include <alloc.h>

void *malloc (size_t num_bytes);
```

O parâmetro *num_bytes* especifica o número de bytes que você quer para o tamanho da matriz. Se a função *malloc* alocar com sucesso o intervalo de byte, *malloc* retornará um ponteiro para o início do intervalo. Se

um erro ocorrer, *malloc* retornará *NULL*. O programa a seguir, *malloc.c*, usa *malloc* para alocar memória para uma matriz de string de caracteres, uma matriz de valores inteiros, e uma matriz de ponto flutuante:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
   char *string;
   int *int_valores;
   float *float_valores;

   if ((string = (char *) malloc(50)))
     printf("String de 50 bytes alocada com sucesso\n");
   else
     printf("Erro ao alocar a string\n");
   if ((int_valores = (int *) malloc(100 * sizeof(int))) != NULL)
     printf("Aloquei com sucesso int_valores[100]\n");
   else
     printf("Erro ao alocar int_valores[100]\n");
   if ((float_valores = (float *) malloc(25 * sizeof(float))) != NULL)
     printf("Aloquei com sucesso float_valores[25]\n");
   else
     printf("Erro ao alocar float_valores[25]\n");
  }
```

Como você pode ver, o programa chama *malloc* com o número de bytes requerido. Se *malloc* retornar *NULL*, o programa exibirá uma mensagem de erro.

REVISITANDO A CONVERSÃO 594

Na dica anterior vimos que é possível usar a função da biblioteca malloc para instruir seus programas a alocar memória durante a execução. Como você aprendeu, a função *malloc* retorna um ponteiro *void*, como mostrado aqui:

```
void *malloc(size_t num_bytes);
```

Quando você usa *malloc* para alocar memória, seus programas devem converter o resultado de *malloc* para um ponteiro do tipo que você quer. Por exemplo, o comando a seguir usa *malloc* para alocar um ponteiro para 100 valores do tipo *int*:

```
int *int_valores;

int_valores = (int *) malloc(100 * sizeof(int));
```

Se você estiver alocando memória para conter 50 valores em ponto flutuante, seus comandos se tornarão os seguintes:

```
float *float valores;

float_valores  = (float *) malloc(50 * sizeof(float));
```

Quando você converte o valor de retorno de *malloc* desse modo, pode eliminar as mensagens de advertência do compilador.

LIBERANDO A MEMÓRIA QUANDO ELA NÃO É MAIS NECESSÁRIA 595

Como você aprendeu, seus programas podem usar a função *malloc* para alocar memória durante a execução para armazenar matrizes e outros itens. Quando seu programa não precisa mais da memória, ele deve liberar a memória para poder usá-la para outro propósito. Para liberar a memória alocada, seus programas podem usar a função *free*, como mostrado aqui:

```
#include <alloc.h>

void free (void  *ptr);
```

O parâmetro *ptr* é um ponteiro para o início do intervalo de memória que você quer liberar. O programa a seguir, *libera.c*, usa *malloc* para alocar uma matriz de inteiros. O programa então usa a matriz. Quando o programa não precisa mais da matriz, ele usa a função *free* para liberar a memória que corresponde à matriz, como mostrado aqui:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
   int *int_valores;
   int i;

   if ((int_valores = malloc(100 * sizeof(int))) == NULL)
     printf("Erro ao alocar a matriz\n");
   else
     {
       for (i = 0; i < 100; i++)
          int_valores[i] = i;
       for (i = 0; i < 100; i++)
          printf("%d ", int_valores[i]);
       free(int_valores);
     }
 }
```

Nota: Se seus programas não usarem free para liberar a memória, o programa automaticamente liberará a memória quando terminar. No entanto, como regra, seus programas deverão liberar a memória assim que não precisarem mais dela.

596 ALOCANDO MEMÓRIA USANDO A FUNÇÃO CALLOC

Como você aprendeu, seus programas podem usar a função *malloc* para alocar memória dinamicamente durante a execução. Quando você usa *malloc*, especifica o número de bytes que quer alocar. Além de usar *malloc*, C permite que seus programas aloquem memória usando *calloc*. A diferença entre as duas funções é que, com *malloc*, você especifica o número de bytes que quer; enquanto que, com *calloc*, você especifica o número de elementos de um tamanho específico que quer, como mostrado aqui:

```
#include <alloc.h>

void *calloc(size_t num_items, size_t tamanho_item);
```

O parâmetro *num_bytes* especifica para quantos elementos *calloc* precisa alocar memória. O parâmetro *tamanho_item* especifica o tamanho em bytes de cada elemento. Se *calloc* alocar a memória com sucesso, *calloc* retornará um ponteiro para o início do intervalo de memória. Se um erro ocorrer, *calloc* retornará *NULL*. O programa a seguir, *calloc.c*, usa *calloc* para alocar vários tipos diferentes de matrizes:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
   char *string;
   int *int_valores;
   float *float_valores;

   if ((string = (char *) calloc(50, sizeof(char))))
     printf('String de 50 bytes alocada com sucesso\n");
   else
     printf('Erro ao alocar a string\n");
   if ((int_valores = (int *) calloc(100, sizeof(int))) != NULL)
```

```
    printf("Aloquei com sucesso int_valores[100]\n");
  else
    printf("Erro ao alocar int_valores[100]\n");
  if ((float_valores = (float *) calloc(25, sizeof(float))) != NULL)
    printf("Aloquei com sucesso float_valores[25]\n");
  else
    printf("Erro ao alocar float_valores[25]\n");
}
```

Nota: Quando seu programa termina de usar a memória alocada por calloc, deve usar free para liberá-la.

COMPREENDENDO O HEAP 597

Quando seus programas alocam memória dinamicamente, a biblioteca de execução de C recebe a memória a partir de um reservatório de memória não-usado chamado *heap*. Quando você compila programas usando o modelo de memória small, o heap é a área de memória entre o topo da área de dados do seu programa e a pilha, como mostrado na Figura 597.

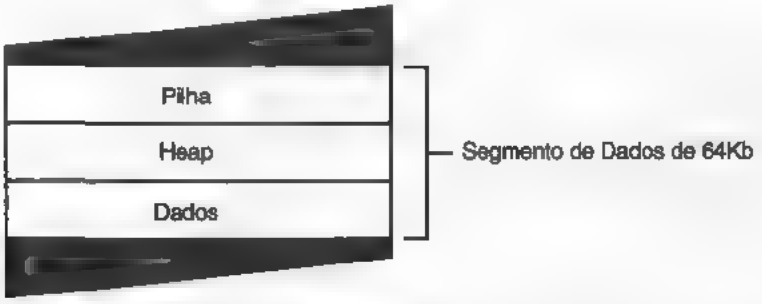

Figura 597 O heap reside entre a área de dados do programa e a pilha.

Como você pode ver, o heap reside no segmento de dados do seu programa. Portanto, a quantidade de espaço de heap disponível para seu programa é fixo para esse programa, mas pode variar de um programa para outro. Quando você usa *calloc* ou *malloc* para alocar memória, o máximo de memória que as funções podem alocar é 64Kb (assumindo que o heap não contenha dados e nenhuma pilha). O programa a seguir, *semespac.c*, tenta alocar três matrizes de 30Kb. Como o heap não tem 90Kb disponíveis, a alocação de memória falha, como mostrado aqui:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
  char *um, *dois, *tres;

  if ((um = (char *) malloc(30000)) == NULL)
    printf("Erro ao alocar matriz um\n");
  else if ((dois = (char *) malloc(30000)) == NULL)
    printf("Erro ao alocar matriz dois\n");
  else if ((tres = (char *) malloc(30000)) == NULL)
    printf("Erro ao alocar matriz tres\n");
  else
    printf("Todas as matrizes alocadas com sucesso\n");
  }
```

No modelo de memória large, o tamanho total do heap não está restrito a 64Kb; no entanto, o maior valor que você pode alocar a qualquer tempo ainda está restrito a um segmento de 64Kb. Para alocar um valor maior que 64Kb, você precisa usar o modelo huge. Experimente compilar o programa *semespac.c* usando o modelo de memória large. O programa deve então poder satisfazer os requisitos de alocação de memória.

Os programas Windows usam o heap de forma similar aos programas do DOS. No entanto, os programas Windows têm acesso a dois heaps: *heap global* e o *heap local*. Todos os programas podem usar o heap global, que o Windows usa para tratar os blocos de memória alta (256 bytes ou mais). O Windows também dá a cada pro-

grama acesso a seu próprio heap local, que o Windows usa para tratar blocos de memória pequenos (256 bytes ou menos). Como regra, a maioria dos programas Windows trabalha a partir do heap global — porque ele não tem uma limitação de tamanho efetiva. No entanto, seus programas podem usar o heap local para propósitos de armazenamento de pouca memória.

598 CONTORNANDO O LIMITE DE 64KB PARA O HEAP

Como você aprendeu, quando seus programas baseados no DOS alocam memória a partir do heap, eles podem alocar no máximo 64Kb de memória. Como o limite de 64Kb é uma restrição do DOS (PC no modo real), muitos compiladores baseados no DOS fornecem funções chamadas *farmalloc* e *farcalloc* que permitem a seus programas alocar memória a partir de um *heap far*, que reside fora do segmento de dado atual, como mostrado aqui:

```
#include <alloc.h>

void far *farcalloc(unsigned long num_itens, unsigned long tamanho_elemento);
void far *farmalloc(unsigned long num_bytes);
```

Os parâmetros que seus programas passam para *farcalloc* e *farmalloc* são idênticos em função àqueles passados para *calloc* e *malloc*. Quando você alocar memória a partir do heap far, usará um ponteiro *far* para os dados. O programa a seguir, *fmalloc.c*, aloca várias matrizes a partir do heap far:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
  {
   char far *string;
   int far *int_valores;
   float far *float_valores;

   if ((string = (char *) farmalloc(50)))
     printf("String de 50 bytes alocada com sucesso\n");
   else
     printf("Erro ao alocar string\n");
   if ((int_valores = (int *) farmalloc(100 * sizeof(int))) != NULL)
     printf("int_valores[100] alocada com sucesso\n");
   else
     printf("Erro ao alocar int_valores[100]\n");
   if ((float_valores = (float *) farmalloc(25 * sizeof(float))) != NULL)
     printf("float_valores[25] alocada com sucesso\n");
   else
     printf("Erro ao alocar float_valores[25]\n");
 }
```

No modelo de memória large, seu programa trata todos os ponteiros como ponteiros far. No entanto, o uso de ponteiros far é transparente dentro de seu aplicativo.

Nota: Quando seus programas usam farcalloc ou farmalloc para alocar memória a partir do heap far, você usaria a função farfree para liberar a memória quando seus programas não precisarem mais dela. Se você estiver usando um compilador diferente, os nomes dessas funções poderão ser diferentes. Consulte a descrição do heap far na documentação do seu compilador.

Dentro do Windows, você contornará a restrição do tamanho do heap alocando a partir do heap global, em vez do heap local. Na verdade, a maioria dos compiladores Windows aloca a partir do heap global por padrão e fornece um comando diferente que você deve usar para alocar a partir do heap local.

599 ALOCANDO MEMÓRIA A PARTIR DA PILHA

Como você aprendeu, as funções *malloc* e *calloc* lhe permitem alocar memória a partir do heap. Quando você tiver terminado de usar a memória, deverá liberá-la usando *free*. Dependendo do seu programa, algumas vezes você precisa alocar memória que somente existe durante uma chamada de função específica. Para fazer isso, seus programas podem usar *alloca* para alocar a memória a partir da pilha, como mostrado aqui:

```
#include <malloc.h>

void *alloca(size_t num_bytes);
```

O parâmetro *num_bytes* especifica o tamanho do intervalo de memória que seu programa precisa alocar. Se *alloca* for bem-sucedida, ela retornará um ponteiro para o início do bloco de memória. Se um erro ocorrer, a função retornará *NULL*. Não use *free* para liberar memória que seu programa tenha alocado usando *alloca* — *free* trabalha com o heap, enquanto que *alloca* trabalha com a pilha. O programa liberará a memória que alocou automaticamente quando a função que contém a memória alocada terminar.

Nota: Para o programa restaurar o ponteiro da pilha corretamente, a função precisa conter variáveis locais. Para garantir uma estrutura da pilha correta, declare uma variável local após declarar a variável ponteiro para a qual alloca atribui seu resultado, como mostrado aqui:

```
char *ponteiro;
char pilha_acerto[1];

pilha_acerto[0] = NULL;
ponteiro = alloca(tamanho);
```

O programa a seguir, *alloca.c*, ilustra como usar a função *alloca:*

```
#include <stdio.h>
#include <malloc.h>

void alguma_funcao(size_t tamanho)
 {
  int i;
  char *ponteiro;
  char pilha_acerto[1];
  pilha_acerto[0] = NULL;
  if ((ponteiro = alloca(tamanho)) == NULL)
    printf("Erro ao alocar %u bytes da pilha\n", tamanho);
  else
   {
     for (i = 0; i < tamanho; i++)
       ponteiro[i] = i;
     printf("Um buffer de %u bytes foi alocado e usado\n", tamanho);
   }
 }

void main(void)
 {
  alguma_funcao(1000);
  alguma_funcao(32000);
  alguma_funcao(65000);
 }
```

ALOCANDO DADOS HUGE 600

Já vimos que, o tamanho da maior matriz que você pode criar é 64Kb. Se seu aplicativo precisar de uma matriz maior, você poderá alocar memória para uma matriz *huge*. Para ajudar seus programas a trabalhar com estruturas de dados huge, muitos compiladores baseados no MS-DOS oferecem as funções *halloc* e *hfree*, como mostrado aqui:

```
#include <malloc.h>

void huge *halloc(long num_elementos, size_t tamanho);
void hfree(void huge *ponteiro);
```

O parâmetro *num_bytes* especifica o número de elementos na matriz. O parâmetro *tamanho* especifica o tamanho em bytes de cada elemento. Se *halloc* for bem-sucedida, ela retornará um ponteiro para o início da área de memória. Se ocorrer um erro, *halloc* retornará *NULL*. O programa a seguir, *hugeint.c*, usa *halloc* para alocar uma matriz de 100.000 bytes:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
 {
   long int i;
   int huge *matriz_grande;

   if ((matriz_grande = (int huge *) halloc (100000L,
         sizeof(long int))) == NULL)
      printf ("Erro ao alocar a matriz huge\n");
   else
    {
      printf("Preenchendo a matriz\n");
      for (i = 0; i < 100000L; i++)
         matriz_grande[i] = i % 32768;
      for (i = 0; i < 100000L; i++)
         printf ("%d ", matriz_grande[i]);
      hfree(matriz_grande);
    }
 }
```

Nota: Novamente, como você aprendeu, a limitação no tamanho da matriz não se aplica aos programas que você criar dentro do ambiente Windows. Na verdade, alguns compiladores Windows não permitem mais o uso da palavra-chave huge na declaração de matrizes.

601 ALTERANDO O TAMANHO DE UM BLOCO ALOCADO

Como você aprendeu, C permite que seus programas aloquem memória dinamicamente durante a execução. Após você alocar um bloco de memória, algumas vezes mais tarde precisará alterar o tamanho do bloco. Nesses casos, seus programas poderão usar *realloc*, como mostrado aqui:

```
#include <stdlib.h>

void *realloc(void *bloco, size_t bytes_desejados);
```

O parâmetro *bloco* é um ponteiro para a memória alocada anteriormente. O parâmetro *bytes_desejados* é o tamanho requerido para o novo bloco. A função *realloc* pode reduzir ou expandir um bloco Se *realloc* for bem-sucedida, ela retornará um ponteiro para o bloco, que poderá ser um ponteiro diferente do original. Em outras palavras, *realloc* pode mover os blocos para encontrar espaço (copiando os dados, conforme necessário). Se ocorrer um erro, *realloc* retornará *NULL*. O programa a seguir, *realloc.c*, usa *realloc* para aumentar o tamanho de um bloco de 100 para 1.000 bytes:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
{
   char *string, *nova_string;

   if ((string = (char *) malloc(100)))
   {
     printf("String de 100 bytes alocada com sucesso\n");
     if ((nova_string = (char *) realloc(string, 1000)))
       printf("Tamanho da string aumentado para 1000\n");
     else
       printf("Erro ao realocar a string\n");
   }
  else
     printf("Erro ao alocar a string  de 100 bytes\n");
}
```

COMPREENDENDO BRK 602

Como você aprendeu, o heap inicia na posição que segue imediatamente o último byte no segmento de dados. O valor *break* é o endereço em que o heap inicia. A função *brk* permite que seus programas alterem o valor de break, atribuindo a ele um endereço específico, como mostrado aqui:

```
#include <alloc.h>

int brk(void *endereco);
```

Se a função *brk* for bem sucedida, ela retornará o valor 0. Se um erro ocorrer, *brk* retornará -1. O programa a seguir, *brk.c*, usa *brk* para definir o valor de break 512 bytes antes de sua posição atual. O programa usa a função *coreleft* para exibir a quantidade de heap disponível antes e após a operação *brk*, como mostrado aqui:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   char *ptr;

   printf("Heap inicial disponível %u\n", coreleft());
   ptr = malloc(1); // Pega o ponteiro para valor de quebra atual
   if (brk(ptr + 512) == 0)
   printf("Heap final disponível %u\n", coreleft());
 }
```

VALIDANDO O HEAP 603

Caso esteja experimentando erros em um programa que aloca memória dinamicamente e não possa localizar a origem do erro, então você poderia considerar efetuar *validações de heap*. Para lhe ajudar a testar o estado do heap, muitos compiladores fornecem uma coleção de rotinas de bibliotecas de execução, tais como *heapwalk* e *heapcheck*. Várias dicas a seguir apresentam modos de seus programas testarem o heap.

EFETUANDO UMA VERIFICAÇÃO RÁPIDA DO HEAP 604

Como você aprendeu, para lhe ajudar a localizar erros nos seus programas que efetuam alocação dinâmica de memória, você poderia querer verificar o estado do heap. Uma rotina que seus programas podem usar para verificar o heap é *heapcheck*, como mostrado aqui:

```
#include <alloc.h>

int heapcheck(void);
```

A função *heapcheck* percorre o heap e examina cada uma de suas entradas. A função retorna um dos valores listados na Tabela 604.

Tabela 604 Os valores de retorno para a função ***heapcheck***.

Valor	Descrição
_HEAPEMPTY	Sem heap
_HEAPOK	O heap é verificado
_HEAPCORRUPT	Uma ou mais entradas corrompidas

O programa a seguir, *heapchk.c*, usa a função *heapcheck* para testar o estado do heap:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
```

```
  char *buffer, *segundo_buffer;
  int i, estado;

  buffer = malloc(100);
  segundo_buffer = malloc(100);
  estado = heapcheck();
  if (estado == _HEAPOK)
    printf("O heap está ok\n");
  else if (estado ==  HEAPCORRUPT)
     printf("O heap está corrompido\n");
  for (i = 0; i <= 100; i++)
     buffer[i] = i;
  estado = heapcheck();
  if (estado == _HEAPOK)
     printf("O heap está ok\n");
  else if (estado == _HEAPCORRUPT)
     printf("O heap está corrompido\n");
}
```

Quando o programa aloca a memória, *heapcheck* retorna um valor de status dizendo que o heap está OK. No entanto, após o programa atribuir valores ao heap, *heapcheck* retorna um valor de status dizendo que o heap está corrompido. Se você examinar o laço *for* atentamente, verá que ele atribui 101 valores a um buffer de 100 bytes (o que corrompe a entrada). Usando *heapcheck*, você pode detectar esses erros muito rapidamente.

605 PREENCHENDO O ESPAÇO LIVRE DO HEAP

Um modo de detectar erros no uso de memória em programas que trabalham com memória dinâmica é preencher todo o espaço livre do heap com um valor específico. Em seguida, à medida que você executar operações de memória, poderá testar se o valor foi modificado. Para lhe ajudar a testar o espaço livre no heap, muitos compiladores C fornecem as seguintes funções:

```
#include <alloc.h>

int heapcheckfree(unsigned int valor);
int heapfillfree(unsigned int valor);
```

O parâmetro *valor* é o valor que você quer atribuir ao espaço livre no heap. As funções retornam um dos valores listados na Tabela 605.

Tabela 605 Os valores de retorno para ***heapcheckfree*** e ***heapfillfree***.

Valor	Descrição
_HEAPEMPTY	Sem heap
_HEAPOK	O heap é verificado
_HEAPCORRUPT	Uma ou mais entradas corrompidas
_BADVALUE	Um valor diferente foi encontrado

O programa a seguir, *encheap.c*, usa *heapcheckfree* e *heapfillfree* para detectar um erro de programação:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   char *buffer1, *buffer2, *buffer3;
   int i, estado;

   buffer1 = malloc(100);
   buffer2 = malloc(200);
   buffer3 = malloc(300);
   free(buffer2);       // Espaço livre no meio
```

```
  estado = heapfillfree('A');
  if (estado == _HEAPOK)
    printf("O heap está ok\n");
  else if (estado ==  HEAPCORRUPT)
    printf("O heap está corrompido\n");
  for (i = 0; i <= 150; i++)
    buffer1[i] = i;
  estado = heapcheckfree('A');
  if (estado == _HEAPOK)
     printf("O heap está ok\n");
  else if (estado == _HEAPCORRUPT)
     printf("O heap está corrompido \n");
  else if (estado == _BADVALUE)
     printf("O valor foi alterado no espaço livre\n");
}
```

VERIFICANDO UMA ENTRADA ESPECÍFICA NO HEAP 606

Na Dica 604 você aprendeu como usar a função *heapcheck* para testar o status de todo o heap. À medida que você verificar erros, poderá também querer testar o estado de entradas de heap individual. Para efetuar o teste, seus programas podem usar a função *heapchecknode*, como mostrado aqui:

```
#include <alloc.h>

int heapchecknode(void *bloco);
```

O parâmetro *bloco* é um ponteiro para um bloco de memória alocado dinamicamente. A função retornará um dos valores mostrados na Tabela 606.

Tabela 606 Os valores de retorno para ***heapchecknode***

Valor	Descrição
_HEAPEMPTY	Sem heap
_HEAPOK	O heap é verificado
_HEAPCORRUPT	Um ou mais entradas corrompidas
_BADNODE	O bloco não foi encontrado
_FREEENTRY	O bloco está livre
_USEDENTRY	O bloco está em uso

O programa a seguir, *heapnodo.c*, ilustra como usar a função *heapchecknode*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   char *buffer, *segundo_buffer;
   int i, estado;

   buffer = malloc(100);
   segundo_buffer = malloc(100);
   estado = heapchecknode(buffer);
   if (estado == _USEDENTRY)
      printf("O buffer está ok\n");
   else
      printf("O buffer não está ok\n");
   estado = heapchecknode(segundo buffer);
   if (estado == _USEDENTRY)
      printf("segundo_buffer está ok\n");
   else
      printf("segundo_buffer não está ok\n");
```

```
  for (i = 0; i <= 100; i++)
     buffer[i] = i;
  estado = heapchecknode(buffer);
  if (estado == _USEDENTRY)
     printf("O buffer está ok\n");
  else
     printf("O buffer não está ok\n");
  estado = heapchecknode(segundo_buffer);
  if (estado == _USEDENTRY)
     printf("segundo_buffer está ok\n");
  printf("segundo_buffer não está ok\n");
}
```

607 PERCORRENDO AS ENTRADAS DO HEAP

Para lhe ajudar a examinar as entradas individuais do heap, muitos compiladores fornecem uma função chamada *heapwalk*, que lhe permite exibir o tamanho e o estado (em uso ou disponível) para cada entrada no heap, como mostrado aqui:

```
#include <alloc.c>

int heapwalk(struct heapinfo *info);
```

O parâmetro *info* é um ponteiro para uma estrutura do tipo *heapinfo*, como mostrado aqui:

```
struct heapinfo
 {
   void *pointer;
   unsigned int size;
   int in_use;
 };
```

Antes da primeira chamada a *heapwalk*, você precisa definir o membro *ponteiro* das estrutura *heapinfo* como *NULL*. A função *heapwalk* retorna um dos valores mostrados na Tabela 607.

Tabela 607 Os valores de retorno para ***heapwalk***.

Valor	Descrição
_HEAPEMPTY	Nenhum heap
_HEAPOK	O heap é verificado
_HEAPEND	Última entrada no heap

O programa a seguir, *heapwalk.c*, percorre as entradas do heap usando *heapwalk*:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   char *buffer1, *buffer2, *buffer3;
   struct heapinfo node = { NULL, 0, 0};

   buffer1 = malloc(100);
   buffer2 = malloc(200);
   buffer3 = malloc(300);
   free(buffer2);
   while (heapwalk(&node) == _HEAPOK)
     printf("Tamanho %u bytes Estado %s\n", node.size,
       (node.in_use) ? "Em uso ": "Livre");
 }
```

EXAMINANDO O CONTEÚDO DE UMA POSIÇÃO DE MEMÓRIA ESPECÍFICA 608

Dependendo da função do seu programa, algumas vezes você desejará acessar posições específicas de segmento e de deslocamento na memória. Se você estiver trabalhando com ponteiros *far*, poderá combinar um segmento e um deslocamento do endereço usando *MK_FP*. Além disso, seus programas podem usar as funções *peekb* e *peek*, como mostrado aqui:

```
#include <dos.h>

char peekb(unsigned segmento, unsigned desloc);
char peek(unsigned segmento, unsigned desloc);
```

Os parâmetros *segmento* e *deslocamento* combinam para especificar a posição de memória desejada. O programa a seguir, *arq_tela.c*, usa a função *peekb* para capturar o conteúdo atual da tela (modo de texto) e para enviar a captura para o arquivo *salvatel.dat*. O programa lê o caractere e o byte de atributo. Portanto, o programa *arq_tela.c* precisa ler 4.000 caracteres e 4.000 atributos, como mostrado aqui:

```
#include <stdio.h>
#include <dos.h>
#define VIDEO 0xB800   // CGA base

void main(void)
 {
   FILE *pa;
   int desloc;

   if ((pa = fopen("SALVATEL.DAT", "wb")) == NULL)
     printf("Erro ao abrir o arquivo\n");
   else
    {
      for (desloc = 0; desloc < 8000; desloc++)
        fprintf(pa, "%c", peekb(VIDEO, desloc));
      fclose(pa);
    }
 }
```

Nota: O programa arq_tela.c usa o endereço-base do vídeo CGA B800H. Se você estiver usando um monitor EGA, VGA ou outro, você poderá precisar alterar esse endereço-base.

COLOCANDO VALORES NA MEMÓRIA 609

Na dica anterior você aprendeu como usar as funções *peekb* e *peek* para ler valores de endereços específicos de segmento e de deslocamento na memória. De um modo similar, a maioria dos compiladores C fornece as funções *poke* e *pokeb*, que permitem que seus programas coloquem valores em posições específicas na memória, como mostrado aqui:

```
#include <dos.h>

void pokeb (unsigned segmento, unsigned desloc, char valor);
void poke (unsigned segmento, unsigned desloc, char valor);
```

O programa a seguir, *res_tela.c*, usa a função *pokeb* para restaurar o conteúdo da tela que o programa *arq_tela.c* salvou anteriormente:

```
#include <stdio.h>
#include <dos.h>
#define VIDEO 0xB800   // CGA base

void main(void)
 {
```

```
  FILE *pa;
  int desloc;
  char valor;

  if ((pa = fopen("SALVATEL.DAT", "rb")) == NULL)
     printf("Erro ao abrir o arquivo\n");
  else
   {
     for (desloc = 0; desloc < 8000; desloc++)
      {
        fscanf(pa, "%c", &valor);
        pokeb(VIDEO, desloc, valor);
      }
     fclose(pa);
   }
}
```

610 COMPREENDENDO AS PORTAS DO PC

O PC usa duas técnicas para se comunicar com os dispositivos de hardware internos. Primeiro, o PC pode referenciar posições de memória que o dispositivo ou o PC reservou anteriormente para o dispositivo. O termo para as operações de entrada e saída que ocorrem por meio dessas posições de memória é *E/S mapeada em memória*. O PC usa E/S mapeada na memória para escrever no vídeo. Além disso, o PC pode se comunicar com dispositivos de hardware usando *portas*. Você pode visualizar melhor uma porta como um registrador no qual o PC ou o dispositivo pode colocar valores específicos. A Tabela 610 lista os endereços de porta que diferentes dispositivos usam em um sistema EISA.

Tabela 610 Endereços de Porta do PC

Porta	Dispositivo	Porta	Dispositivo
00H-1FH	Controlador de DMA	2EFH-2FFH	COM2
20H-3FH	Controlador de interrupção	300H-31FH	Placas de rede
40H-5FH	Temporizador do sistema	378H-37FH	LPT1
60H-6FH	Teclado	380H-38FH	SDLC
70H-7FH	Relógio de tempo real	390H-39FH	Adaptador Cluster
80H-9FH	Registradores de página do DMA	3B0H-3BFH	Monocromático
A0H-BFH	Controlador de interrupções 2	3C0H-3CFH	EGA
C0H-DFH	Controlador de DMA 2	3D0H-3DFH	CGA
F0H-FFH	Co-processador matemático	3F0H-3F7H	Unidade de disquete
1F0H-1FFH	Disco rígido	3F8H-3FFH	COM1
200H-220H	Adaptador de jogos	400H-4FFH	DMA
270H-27FH	LPT2	500H-7FFH	Apelidos 100H-3FFH
2B0H-2DFH	EGA alternativo	800H-8FFH	CMOS
2E0H-2E7H	COM4	900H-9FFH	Reservado
2E8H-2EFH	COM3	9FFH-FFFFH	Reservado

O significado de cada porta depende do dispositivo correspondente. Para obter os significados específicos das portas, consulte a documentação técnica do PC ou do dispositivo.

611 ACESSANDO OS VALORES DAS PORTAS

Se seus programas controlam o hardware no baixo nível, algumas vezes você precisará ler ou escrever um valor em uma porta. Para que seus programas façam isso, a maioria dos compiladores baseados no DOS fornece as seguintes funções:

```
#include <dos.h>

int inport(int ender_porta);
```

```
char inportb(int ender_porta);
void outport(int ender_porta, int valor);
void outportb(int ender_porta, unsigned char valor);
```

O parâmetro *ender_porta* especifica o endereço da porta desejada, como listado na dica anterior. O parâmetro *valor* especifica o valor de 16 ou de 8 bits que seu programa quer enviar para a porta. A dica a seguir ilustra como usar a função *inportb* para ler e exibir o conteúdo da memória CMOS do PC.

COMPREENDENDO A CMOS 612

Como você sabe, o PC armazena as informações de configuração do sistema na memória CMOS, incluindo seus tipos de unidades, data do sistema e assim por diante. O PC não acessa a CMOS usando o endereçamento padrão de segmento e de deslocamento. Em vez disso, ele usa endereços de porta para se comunicar com a CMOS. Como você aprendeu na dica anterior, a maioria dos compiladores C fornece funções, tais como *inport* e *outport*, para ajudar seus programas a acessar as portas do PC. O programa a seguir, *exibcmos.c*, usa a função *inport* para obter e exibir as informações da CMOS:

```
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

void main(void)
 {
   struct CMOS {
     unsigned char current_second;
     unsigned char alarm_second;
     unsigned char current_minute;
     unsigned char alarm_minute;
     unsigned char current_hour;
     unsigned char alarm_hour;
     unsigned char current_day_of_week;
     unsigned char current_day;
     unsigned char current_month;
     unsigned char current_year;
     unsigned char status_registers[4];
     unsigned char diagnostic_status;
     unsigned char shutdown_code;
     unsigned char drive_types;
     unsigned char reserved_x;
     unsigned char disk_1_type;
     unsigned char reserved;
     unsigned char equipment;
     unsigned char lo_mem_base;
     unsigned char hi_mem_base;
     unsigned char hi_exp_base;
     unsigned char lo_exp_base;
     unsigned char fdisk_0_type;
     unsigned char fdisk_1_type;
     unsigned char reserved_2[19];
     unsigned char hi_check_sum;
     unsigned char lo_check_sum;
     unsigned char lo_actual_exp;
     unsigned char hi_actual_exp;
     unsigned char century;
     unsigned char information;
     unsigned char reserved3[12];
   } cmos;

   char i;
   char *ponteiro;
   char byte;
```

```
  ponteiro = (char *) &cmos;
  for (i = 0; i < 0x34; i++)
   {
     outportb(0x70, i);
     byte = inportb(0x71);
     *ponteiro++ = byte;
   }
  printf("Data atual %d/%d/%d\n", cmos.current_month,
    cmos.current_day, cmos.current_year);
  printf("Horário atual %d:%d:%d\n", cmos.current_hour,
    cmos.current_minute, cmos.current_second);
  printf("Tipo de disco rígido %d\n", cmos.fdisk_0_type);
}
```

613 COMPREENDENDO OS MODELOS DE MEMÓRIA

Quando você cria programas no ambiente do PC, o compilador usa um *modelo de memória* para determinar quanta memória o sistema operacional aloca para seu programa. Como você aprendeu, o PC divide a memória em blocos de 64Kb chamados segmentos. Normalmente, seu programa usa um segmento para código (as instruções do programa) e um segundo segmento para dados. Caso seu programa seja muito grande ou usa grandes quantidades de dados, algumas vezes o compilador precisará fornecer múltiplos segmentos de código e de dados. O modelo de memória define o número de segmentos que o compilador pode usar para cada um. Os modelos de memória são importantes porque, se você usa o modelo de memória errado, seu programa pode não ter memória suficiente para executar.

Normalmente, o compilador selecionará um modelo de memória que é grande o suficiente para seu programa. No entanto, como você aprenderá, quanto maior for o modelo de memória, mais lenta será a execução do programa. Portanto, seu objetivo é sempre usar o menor modelo de memória que satisfaça as necessidades do seu programa. A maioria dos compiladores suporta os modelos tiny, small, medium, compact, large e huge. Várias dicas a seguir descrevem esses modelos de memória em detalhe. Para selecionar um modelo de memória, você normalmente inclui uma opção na linha de comando do compilador. Consulte a documentação de seu compilador para saber quais são as opções de modelo de memória.

Nota: Como vimos, os diferentes tipos de modelo de memória que você pode usar para criar programas C/C++ em um ambiente DOS não se aplicam ao ambiente Windows, que somente usa o modelo de memória virtual. Embora as próximas sete dicas sejam úteis caso você pretenda ou não desenvolver programas para o DOS, elas servem unicamente como informativas para os programadores Windows.

614 COMPREENDENDO O MODELO DE MEMÓRIA TINY

Um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. O menor e o mais rápido modelo de memória é o *tiny*. Devido a sua natureza compacta, o modelo tiny consome a mínima quantidade de memória e carrega mais rápido que os outros modelos. Como mostra a Figura 614, o modelo tiny combina o código e os dados do programa em um único segmento de 64Kb.

Figura 614 O modelo de memória tiny coloca o código e os dados do programa em um segmento de 64Kb.

Se você estiver criando programas pequenos, tais como muitos dos programas de exemplo apresentados neste livro, deverá instruir o compilador a usar o modelo de memória tiny.

COMPREENDENDO O MODELO DE MEMÓRIA SMALL 615

Um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. O modelo de memória mais comum é o *small*. Como mostra a Figura 615, o modelo small usa um segmento de 64Kb para o código do seu programa e um segundo para os seus dados.

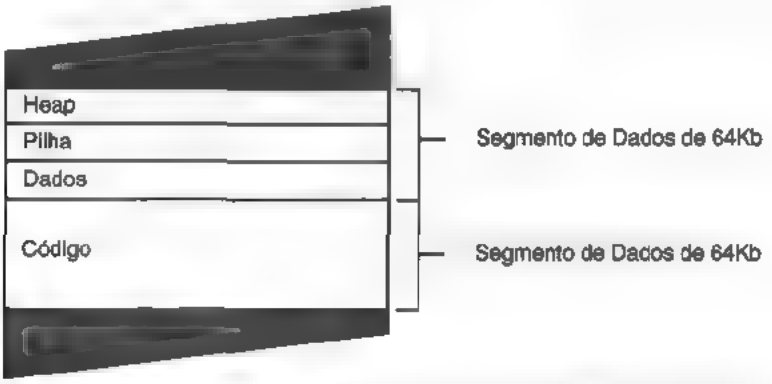

Figura 615 O modelo de memória small usa um segmento de 64Kb para o código do programa e outro para os dados.

A vantagem em usar o modelo de memória small é que todas as chamadas de funções e todas as referências de dados usam endereços near de 16 bits. O programa que usa o modelo small executa mais rápido que um programa equivalente nos modelos de memória maiores.

COMPREENDENDO O MODELO DE MEMÓRIA MEDIUM 616

Um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. Caso seu programa requeira mais de 64Kb de memória para o código, mas somente 64Kb (ou menos) para os dados, então seus programas poderão usar o modelo de memória *medium*. Como mostra a Figura 616, o modelo de memória medium aloca múltiplos segmentos de código e somente um segmento de dados.

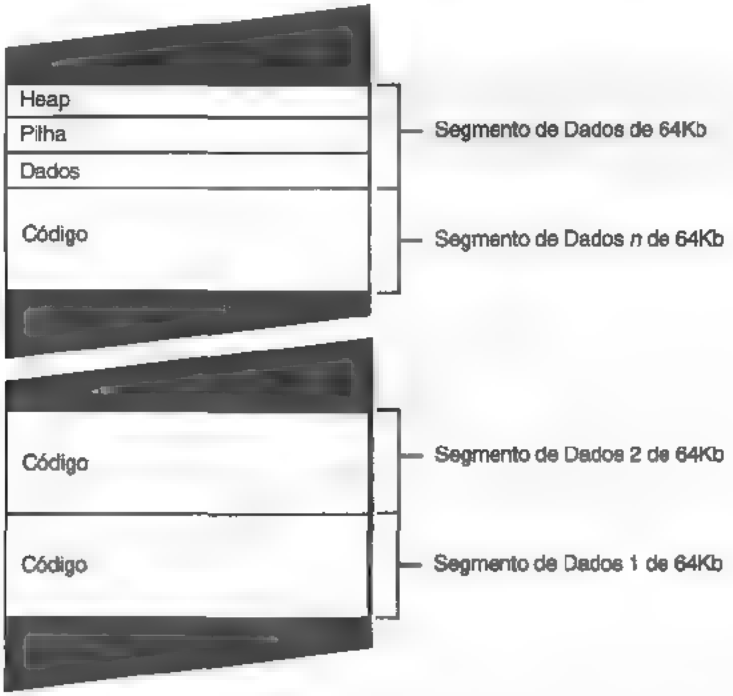

Figura 616 O modelo de memória medium aloca múltiplos segmentos de código e um segmento de dados.

Caso seu programa contenha um grande número de instruções, então o modelo de memória medium oferece acesso de dados rápidos, porque todas as referências de dados usam endereços near. No entanto, como o modelo medium usa múltiplos segmentos de código, todas as chamadas de funções requerem endereços far de 32 bits. A colocação e retirada de endereços de segmentos adicionais na/da pilha degrada um pouquinho o desempenho do programa.

617 COMPREENDENDO O MODELO DE MEMÓRIA COMPACT

Como você aprendeu, um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. Se seu programa usa uma grande quantidade de dados mas poucas instruções, seus programas podem usar o modelo compact. Como mostra a Figura 617, o modelo Compact aloca um segmento de 64Kb para o código de seu programa e múltiplos segmentos para os dados.

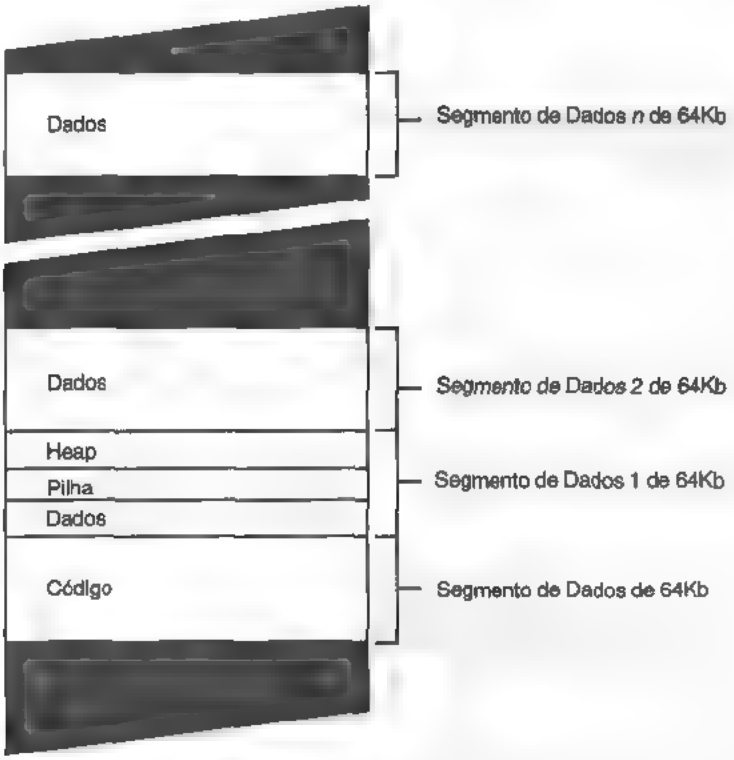

Figura 617 O modelo de memória compact aloca um segmento de 64Kb para código e múltiplos segmentos para os dados.

Como o modelo de memória compact usa um segmento de código, todas as chamadas de funções usam endereços near de 16 bits. Como resultado, as chamadas das funções são mais rápidas que em qualquer modelo de memória maior. As referências de dados, por outro lado, requerem um endereço de segmento e de deslocamento (um endereço far de 32 bits). A sobrecarga necessária para trabalhar com os endereços de deslocamento e de segmento da referência dos dados reduzirá o desempenho do seu programa.

618 COMPREENDENDO O MODELO DE MEMÓRIA LARGE

Um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. Se seu programa contém uma grande quantidade de código e de dados, seu programa pode usar o modelo de memória *large*. Como mostra a Figura 618, o modelo Large aloca múltiplos segmentos de dados e de código.

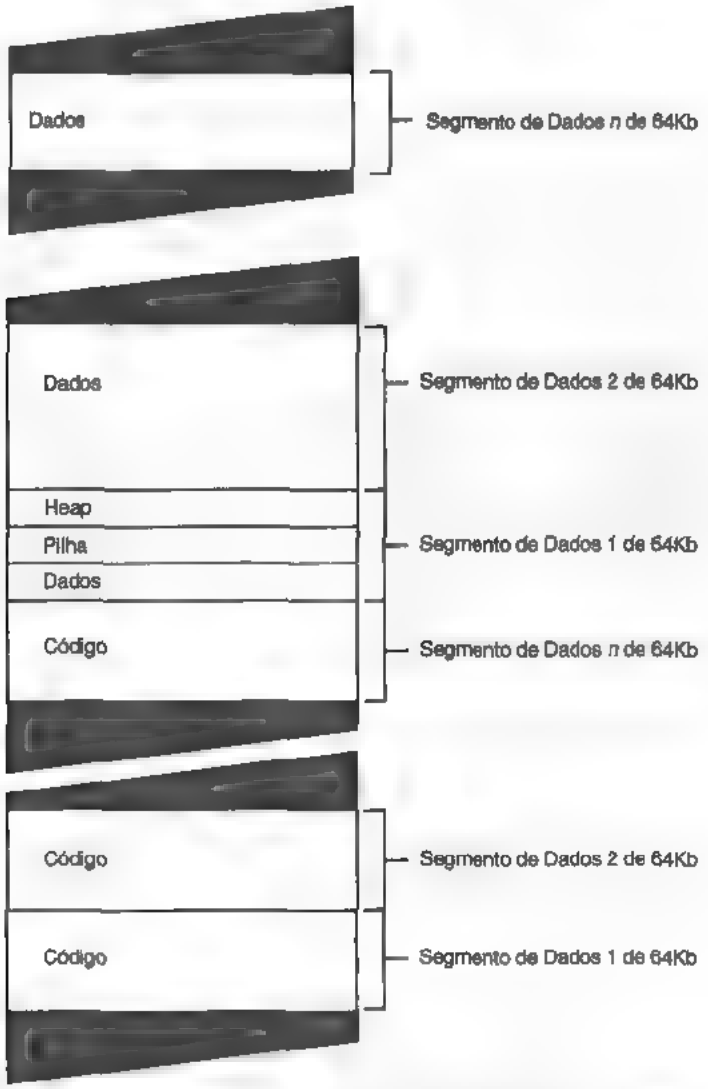

Figura 618 O modelo de memória large aloca múltiplos segmentos de dados e de código

Você deve usar o modelo de memória large somente como um último recurso. Como o modelo de memória large usa múltiplos segmentos de código e de dados, cada chamada de função e cada referência de dados requerem um endereço far de 32 bits. A sobrecarga envolvida com a manipulação de segmento e de deslocamento torna o modelo large o mais lento dos modelos descritos até aqui.

COMPREENDENDO O MODELO DE MEMÓRIA HUGE 619

Um modelo de memória descreve o número de segmentos de memória de 64Kb que o compilador aloca para um programa. Como você aprendeu, a maioria dos compiladores baseados no PC fornece diferentes modelos de memória para satisfazer os requisitos de código e dados do seu programa. No entanto, uma condição especial surge quando seu programa usa uma matriz maior que 64Kb. Para alocar tal matriz, seu programa precisará usar a palavra-chave *huge* para criar um ponteiro, como mostrado aqui:

```
int huge *matriz_grande;
```

Em seguida, seu programa precisa usar a função *halloc* para alocar a memória. O programa a seguir, *hugeint.c*, originalmente apresentado na Dica 600, usa *halloc* para alocar uma matriz de 400.000 bytes:

```
#include <stdio.h>
#include <malloc.h>

void main(void)
 {
    long int i;
    int huge *matriz_grande;

    if ((matriz_grande = (int huge *) halloc (100000L,
          sizeof(long int))) == NULL)
       printf ("Erro ao alocar a matriz huge\n");
    else
      {
        printf("Preenchendo a matriz\n");
        for (i = 0; i < 100000L; i++)
           matriz_grande[i] = i % 32768;
        for (i = 0; i < 100000L; i++)
           printf ("%d ", matriz_grande[i]);
        hfree(matriz_grande);
      }
 }
```

Quando você compilar e executar um programa usando o modelo de memória huge, a maioria dos compiladores usará endereços far, de 32 bits para o código e para os dados (similar ao modelo de memória large). Como resultado, a execução do programa poderá ser mais lenta do que você deseja.

620 DETERMINANDO O MODELO DE MEMÓRIA ATUAL

Dependendo do processamento do seu programa, algumas vezes você precisa compilar o programa usando um modelo de memória diferente. A maioria dos compiladores C predefine uma constante específica para ajudar seus programas a determinar o modelo de memória atual. A Tabela 620, por exemplo, lista as constantes que os compiladores *Turbo C++ Lite*, Microsoft C e Borland C definem para diferentes modelos de memória.

Tabela 620 Constantes que os compiladores Turbo C++ Lite, Microsoft C e Borland C++ definem para indicar o modelo de memória atual.

Modelo de Memória	Microsoft C	Turbo C++ Lite/Borland C
Small	*M_I86SM*	*__SMALL__*
Medium	*M_I86MM*	*__MEDIUM__*
Compact	*M_I86CM*	*__COMPACT__*
Large	*M_I86LM*	*__LARGE__*

Caso seu programa requeira um modelo de memória específico, o programa poderá testar o modelo, como mostrado aqui:

```
#ifndef __MEDIUM__
  printf("O programa requer o modelo de memória Medium\n");
  exit(1);
#endif
```

621 OBTENDO A DATA E A HORA ATUAIS COMO SEGUNDOS DESDE 1/1/1970

À medida que seus programas forem se tornando mais funcionais, eles freqüentemente precisarão saber a data e a hora atuais. A maioria dos compiladores C fornece várias funções que retornam a data e a hora em diferentes formatos. Uma dessas funções é *time*, que retorna a data e hora atuais como segundos desde 00:00 de 1/1/1970. A função retorna um valor do tipo *time_t*, como mostrado aqui:

```
#include <time.h>

time_t time(time_t *data_hora);
```

Se você não quiser passar um parâmetro para *time*, poderá chamar a função com *NULL*, como mostrado aqui:

```
hora_atual = time(NULL);
```

O programa a seguir, *retardo_5.c*, usa a função *time* para implementar um retardo de 5 segundos:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   time_t hora_atual;
   time_t hora_inicio;

   printf("Prestes a fazer uma pausa de 5 segundos\n");
   time(&hora_inicio); // Pega o tempo inicial em segundos
   do {
       time(&hora_atual);
     } while ((hora_atual - hora_inicio) < 5);
   printf("Acabado\n");
 }
```

CONVERTENDO UMA DATA E UMA HORA DE SEGUNDOS PARA ASCII 622

A dica anterior apresentou a função *time*, que retorna a hora atual como segundos desde 00:00 de 1/1/1970. Usando a função *ctime*, seus programas podem converter os segundos para uma string de caracteres no seguinte formato:

```
"Fri Oct 31 11:30:00 1997\n"
```

O programa a seguir, *ctime.c*, ilustra como usar a função *ctime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   time_t hora_atual;

   time(&hora_atual);  // Pega a hora em segundos
   printf("A data e hora atuais são: %s", ctime(&hora_atual));
 }
```

AJUSTE DO HORÁRIO DE VERÃO 623

Várias funções que esta seção apresenta levam em conta o horário de verão. Para efetuar tal processamento, muitos compiladores declaram uma variável global chamada *daylight*. Se o horário de verão está em vigor, a variável contém o valor 1. Se o horário de verão não estiver em vigor, C definirá a variável como 0. As funções *tzset*, *localtime* e *ftime* controlam o valor da variável. O fragmento de código a seguir usa a variável *daylight* para determinar se o horário de verão ou o horário normal está em vigor:

```
if (daylight)
   printf("O horário de verão está ativo\n");
else
   printf("O horário de verão não está ativo\n");
```

Nota: A função tzset atribui o valor à variável daylight.

624 RETARDANDO DURANTE UM DETERMINADO NÚMERO DE MILISSEGUNDOS

Dependendo do seu programa, algumas vezes você precisará que o programa retarde por um determinado número de milissegundos (1/1000 segundos). Por exemplo, você pode querer exibir uma mensagem na sua tela por alguns segundos, mas continuando a execução do programa sem forçar o usuário a pressionar uma tecla. Para esses casos, muitos compiladores C fornecem a função *delay*. A função retardará durante o número de milissegundos especificado, como mostrado aqui:

```
#include <dos.h>

void delay(unsigned milissegundos);
```

Usando a função *delay*, seus programas podem especificar um período de retardo de até 64.535 milissegundos. O programa a seguir, *usaretar.c*, usa a função *delay* para retardar por cinco segundos (5.000 segundos):

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   printf("Prestes a fazer uma pausa de 5 segundos\n");
   delay(5000);
   printf("Acabado\n");
 }
```

625 DETERMINANDO O TEMPO DE PROCESSAMENTO DO SEU PROGRAMA

Quando você segue passos para aumentar o desempenho do seu programa, pode querer medir o tempo que diferentes partes do seu programa consomem. É possível, então, determinar quais seções do seu programa são as mais demoradas. Como regra, você deve começar a otimizar na seção do seu programa que consome mais tempo do processador. Para lhe ajudar a determinar o tempo de processamento do seu programa, C fornece a função *clock*, que retorna o número de tiques do relógio (que ocorrem 18.2 vezes por segundo), como mostrado aqui:

```
#include <time.h>

clock_t  clock(void);
```

A função *clock* retorna o tempo de processamento do programa em tiques do relógio. Para converter o tempo para segundos, divida o resultado pela constante *CLK_TCK*, que é definido no arquivo de cabeçalho *time.h*. O programa a seguir, *clock.c*, usa a função *clock* para exibir o tempo do processador em segundos:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>     // Contém o protótipo de delay

void main(void)
 {
   clock_t tempo_processador;

   printf("Tempo do processador consumido %ld\n", clock() / (long) CLK_TCK);
   delay(2000);
   printf("Tempo do processador consumido %ld\n", (long) clock() / (long)
CLK_TCK);
   delay(3000);
   printf("Tempo do processador consumido %ld\n", clock() / (long) CLK_TCK);
 }
```

Nota: Caso seu compilador não forneça a função delay, você pode usar um ou mais laços for para implementar um retardo.

COMPARANDO DOIS HORÁRIOS 626

Na Dica 621 você aprendeu como usar a função *time* para obter o número de segundos desde 1/1/1970. Quando você trabalha com horário, seus programas freqüentemente precisam comparar dois ou mais horários. Para comparar horários, seus programas podem usar a função *difftime* de C, que retorna a diferença entre dois horários como um valor em ponto flutuante, como mostrado aqui:

```
float difftime(time_t hora_post, time_t hora_inicial);
```

O programa a seguir, *difftime.c*, usa a função *difftime* para uma pausa até que cinco segundos tenham transcorrido:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   time_t hora_inicial;
   time_t hora_atual;

   time(&hora_inicial);
   printf("Prestes a fazer uma pausa de 5 segundos\n");
   do
     {
       time(&hora_atual);
     } while (difftime(hora_atual, hora_inicial) < 5.0);
   printf("Acabado\n");
 }
```

OBTENDO UMA STRING DE DATA 627

Na Dica 622 você aprendeu como usar a função *ctime* para criar uma string de caracteres que contém a data e a hora. Para usar a função *ctime*, você precisa primeiro chamar a função *time* para obter o número de segundos desde 1/1/1970. Caso você somente queira obter a data atual, seus programas podem usar a função *_strdate*, como mostrado aqui:

```
#include <dos.h>

char *_strdate(char *buffer_dados);
```

O buffer da string de caracteres que você passa para a função *_strdate* precisa ser grande o suficiente para conter nove caracteres (a data de oito caracteres e *NULL*). A função *_strdate* coloca a data no formato *mm/dd/aa*. O programa a seguir, *strdate.c*, usa a função *strdate* para exibir a data atual:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {

   char data[9];
   _strdate(data);
   printf("A data atual é %s\n", data);
 }
```

OBTENDO UMA STRING COM O HORÁRIO 628

Já vimos na Dica 622 como usar a função *ctime* para criar uma string de caracteres que contém a data e a hora. Para usar a função *ctime*, você precisa primeiro chamar a função *time* para obter o número de segundos desde 1/1/1970. Se você somente quiser obter o horário atual, seus programas poderão usar a função *_strtime*, como mostrado aqui:

```
#include <dos.h>

char *_strtime(char *buffer_hora);
```

O buffer da string de caracteres que passa para a função *_strtime* precisa ser grande o suficiente para armazenar nove caracteres (os oito caracteres do horário e *NULL*). A função *_strtime* coloca o horário na forma *hh:mm:ss*. O programa a seguir, *strtime.c*, usa a função *_strtime* para exibir o horário atual:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   char hora[9];

   _strtime(hora);
   printf("A hora atual é %s\n", hora);
 }
```

629 LENDO O TEMPORIZADOR DA BIOS

A BIOS do PC tem um temporizador interno que gera pulsos 18,2 vezes por segundo. BIOS armazena na memória o número de pulsos que ocorreram desde a meia-noite. No passado, muitos programas usaram o temporizador da BIOS para provocar uma pausa nos programas até que um certo número de pulsos tivesse ocorrido. No entanto, como discutido anteriormente, seus programas podem especificar um intervalo de tempo mais refinado usando a função *delay*. O temporizador da BIOS permanece útil para gerar uma semente para um gerador de números aleatórios. Muitos compiladores C fornecem duas funções que lhe permitem controlar o temporizador da BIOS — *biostime* e *_bios_timeofday*. A função *biostime* permite que seus programas acessem o número de pulsos do relógio que ocorreram desde a meia-noite. O formato da função *biostime* é:

```
#include <bios.h>

long biostime(int operacao, long novahora);
```

O parâmetro *operacao* lhe permite especificar se você quer ler ou definir o temporizador da BIOS, como mostrado na Tabela 629.1.

Tabela 629.1 Os valores possíveis para o parâmetro ***operacao***

Valor	Significado
0	Lê o valor atual do temporizador
1	Define o valor do temporizador com o valor em *novahora*

A função retorna o número atual de pulsos do relógio. A função *_bios_timeofday* também lhe permite ler ou definir o temporizador da BIOS, como mostrado aqui:

```
#include <bios.h>

long _bios_timeofday(int operacao, long *pulsos);
```

O parâmetro *operacao* novamente especifica se você quer ler ou definir o temporizador, como mostrado na Tabela 629.2.

Tabela 629.2 As definições possíveis para o parâmetro ***operacao***.

Valor	Significado
_TIME_GETCLOCK	Lê o valor atual do temporizador
_TIME_SETCLOCK	Define o valor do temporizador como o valor em pulsos

A função *_bios_timeofday* retorna o valor que o serviço do temporizador da BIOS armazena dentro do registrador AX. O programa a seguir, *biostime.c*, usa ambas essas funções para ler os pulsos atuais da BIOS:

```
#include <stdio.h>
#include <bios.h>

void main(void)
 {
   long pulsos;

   pulsos = biostime(0, pulsos);
   printf("Pulsos desde a meia-noite %ld\n", pulsos);
   _bios_timeofday(_TIME_GETCLOCK, &pulsos);
   printf("Segundos desde a meia-noite %f\n", pulsos / 18.2);
 }
```

TRABALHANDO COM O HORÁRIO LOCAL 630

Na Dica 621 você aprendeu que a função *time* retorna o horário atual em segundos desde a meia-noite de 1/1/1970. Para tornar o temporizador do sistema mais fácil para seus programas usarem, o compilador C fornece a função *localtime*, que converte o tempo em segundos para uma estrutura do tipo *tm*. A estrutura *tm* está definida no arquivo de cabeçalho *time.h* como mostrado aqui:

```
struct tm
 {
   int tm_sec;      // Segundos de 0 a 59
   int tm_min;      // Minutos de 0 a 59
   int tm_hour;     // Horas de 0 a 24
   int tm_mday;     // Dia de 1 a 31
   int tm_mon;      // Mês de 1 a 12
   int tm_year;     // Ano - 1900
   int tm_wday;     // 0 domingo até 6 sábado
   int tm_yday;     // Dia do ano 1 até 365
   int tm_isdst;    // Diferente de 0 se horário de verão
 };
```

O formato da função *localtime* é:

```
#include <time.h>

struct tm *localtime(const time_t *tempor);
```

A função *localtime* usa as variáveis globais *timezone* e *daylight* para ajustar o horário para sua zona horária atual e levar *daylight* em conta. O programa a seguir, *localtm.c*, ilustra como usar a função *localtime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   struct tm *data_atual;

   time_t segundos;
   time(&segundos);
   data_atual = localtime(&segundos);
   printf("Data atual: %d-%d-%d\n", data_atual->tm_mon+1,
     data_atual->tm_mday, data_atual->tm_year);
   printf("Hora atual: %02d:%02d\n", data_atual->tm_hour,
     data_atual->tm_min);
 }
```

TRABALHANDO COM O HORÁRIO DE GREENWICH 631

Na Dica 621 você aprendeu que a função *time* retorna o horário atual em segundos desde a meia-noite de 1/1/1970. Se você trabalhar com usuários internacionais, algumas vezes precisará trabalhar em termos de Horá-

rio de Greenwich (GMT). Para que você possa trabalhar com o Horário de Greenwich, o compilador C oferece a função *gmtime*, que converte o horário em segundos para uma estrutura do tipo *tm*, como mostrado na Dica 630. O formato da função *gmtime* é:

```
#include <time.h>

struct tm *gmtime(const time_t *tempor);
```

A função gmtime usa a variável global daylight para levar em conta o horário de verão. O programa a seguir, gmtime.c, ilustra como usar a função *gmtime:*

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   struct tm *gm_data;
   time_t segundos;
   time(&segundos);
   gm_data = gmtime(&segundos);
   printf("Data atual: %d-%d-%d\n", gm_data->tm_mon+1,
      gm_data->tm_mday, gm_data->tm_year);
   printf("Horário atual: %02d:%02d\n", gm_data->tm_hour,
      gm_data->tm_min);
 }
```

632 OBTENDO O HORÁRIO DO SISTEMA DOS

Se você estiver usando o DOS, seus programas poderão usar a função *gettime* para obter o horário do DOS. A função *gettime* atribui o horário atual para uma estrutura do tipo *time*, que está definida dentro do arquivo de cabeçalho *dos.h* e mostrada aqui:

```
struct time
 {
   unsigned char ti_min;   // Minutos de 0 a 59
   unsigned char ti_hour;  // Horas de 0 a 59
   unsigned char ti_hund;  // Centésimos de seg de 0 a 99
   unsigned char ti_sec;   // Segundos de 0 a 59
 };
```

O formato da função *gettime* é como segue:

```
#include <dos.h>

void gettime(struct time *hora_atual);
```

O programa a seguir, *horados.c*, usa a função *gettime* para obter e depois exibir a hora atual do sistema:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct time hora_atual;
   gettime(&hora_atual);
   printf("Horário atual %02d:%02d:%02d.%d\n", hora_atual.ti_hour,
       hora_atual.ti_min, hora_atual.ti_sec, hora_atual.ti_hund);
 }
```

Nota: Muitos compiladores baseados no DOS também fornecem a função ***_dos_gettime****, que retorna uma estrutura do tipo* ***dostime_t****, como mostrado aqui:*

```
struct dostime_t
 {
```

```
  unsigned char hour;       // 0 a 23
  unsigned char minute;     // 0 a 59
  unsigned char second;     // 0 a 59
  unsigned char hsecond;    // 0 a 99
};
```

O formato da função *_dos_gettime* é como segue:

```
#include <dos.h>

void _dos_gettime(struct dostime_t *hora_atual);
```

*Nota: O CD-ROM que acompanha este livro inclui o programa **wintime.cpp**, que usa a API do Windows para obter a hora atual do sistema.*

OBTENDO A DATA DO SISTEMA DO DOS 633

Se você estiver usando o DOS, seus programas poderão usar a função *getdate* para obter a data do sistema do DOS. A função atribui a data atual para uma estrutura do tipo *date*, que está definida dentro do arquivo de cabeçalho *dos.h* e mostrada aqui:

```
struct date
{
  int da_year;     // Ano atual
  int da_day;      // Dia atual de 1 a 31
  int da_mon;      // Mês atual de 1 a  12
};
```

O formato da função *getdate* é:

```
#include <dos.h>

void getdate(struct date *data_atual);
```

O programa a seguir, *datados.c*, usa a função *getdate* para obter e depois exibir a data atual do sistema:

```
#include <stdio.h>
#include <dos.h>

void main(void)
{
  struct date data_atual;
  getdate(&data_atual);
  printf("Data atual: %d-%d-%d\n", data_atual.da_day,
    data_atual.da_mon, data_atual.da_year);
}
```

*Nota: Muitos compiladores baseados no DOS também fornecem a função **_dos_getdate**, que retorna uma estrutura do tipo **dosdate_t**, como mostrado aqui:*

```
struct dosdate_t
{
  unsigned char day;         // 1 a 31
  unsigned char month;       // 1 a 12
  unsigned int year;        // 1980-2099
  unsigned char dayofweek;  // 0 domingo até 6 sábado
};
```

O formato da função *_dos_getdate* é como segue:

```
#include <dos.h>

void _dos_getdate(struct dosdate_t *data_atual);
```

*Nota: O CD-ROM que acompanha este livro inclui o programa **windate.cpp**, que usa a API do Windows para obter a data atual do sistema.*

634 DEFININDO O HORÁRIO DO SISTEMA DOS

Se você estiver usando o DOS, seus programas poderão usar a função *settime* para definir a hora do sistema DOS exatamente como se você tivesse usado o comando TIME do DOS. Para usar *settime*, você atribui o horário que quer para uma estrutura do tipo *time*, como mostrado na Dica 632. O formato da função *settime* é como segue:

```
#include <dos.h>

void settime(struct time *hora_atual);
```

O programa a seguir, *settime.c*, usa a função *settime* para definir a hora atual do sistema para 12h30min:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct time hora_desejada;

   hora_desejada.ti_hour = 12;
   hora_desejada.ti_min = 30;
   settime(&hora_desejada);
 }
```

Nota: Muitos compiladores baseados no DOS também fornecem a função _dos_settime, que usa uma estrutura do tipo dostime_t para definir a hora do sistema, como mostrado na Dica 632. O formato da função _dos_settime é como segue:

```
#include <dos.h>

void _dos_settime(struct dostime_t *hora_atual);
```

Nota: O CD-ROM que acompanha este livro inclui o programa wsettim.cpp, que usa a API do Windows para obter a hora atual do sistema.

635 DEFININDO A DATA DO SISTEMA DO DOS

Se você estiver usando o DOS, seus programas poderão usar a função *setdate* para obter a data do sistema do DOS. Antes de chamar a função *setdate*, atribua a hora que você quer para uma estrutura do tipo *date*, como mostrado na Dica 633. Em seguida, use um ponteiro para a estrutura para chamar a função. O formato da função *setdate* é como segue:

```
#include <dos.h>

void setdate(struct date *data_atual);
```

O programa a seguir, *defdata.c*, usa a função *setdate* para definir a data atual do sistema como 31 de outubro de 1997:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   struct date data_desejada;

   data_desejada.da_mon = 10;
   data_desejada.da_day = 31;
   data_desejada.da_year = 1997;
   setdate(&data_desejada);
 }
```

Nota: Muitos compiladores baseados no DOS também oferecem a função _dos_setdate, que usa uma estrutura do tipo dosdate_t para definir o sistema, como mostrado na Dica 633. O formato da função _dos_setdate é como segue:

```
#include  <dos.h>
unsigned _dos_setdate(struct dosdate_t *date);
```

Nota: O CD-ROM que acompanha este livro inclui o programa Winsdat.cpp, que usa a API do Windows para definir a data atual do sistema.

CONVERTENDO UMA DATA DO DOS PARA O FORMATO DO UNIX 636

Na Dica 633 você aprendeu como usar a função *getdate* para obter a data do sistema do DOS. Da mesma forma, na Dica 632 você aprendeu como usar a função *gettime* para obter a data do sistema do DOS. Se você estiver trabalhando em um ambiente onde usa tanto o DOS quanto o UNIX, então algumas vezes precisará converter um formato de data e hora baseados no DOS para o formato de data e hora que o UNIX utiliza. Nesses casos, seus programas podem usar a função *dostounix* para efetuar a conversão. A função *dostounix* converte estruturas do tipo *date* e do tipo *time* para segundos desde a meia-noite de 1/1/1970, como mostrado aqui:

```
#include <dos.h>

long  dostounix(struct date *DOS_data, struct time *DOS_hora);
```

O programa a seguir, *dosunix.c*, usa a função *dostounix* para converter a data e a hora atuais do DOS para o formato correspondente do Unix:

```
#include <stdio.h>
#include <dos.h>
#include <time.h>

void main(void)
 {
   struct time hora_dos;
   struct date data_dos;

   time_t formato_unix;
   struct tm *local;
   getdate(&data_dos);
   gettime(&hora_dos);
   formato_unix = dostounix(&data_dos, &hora_dos);
   local = localtime(&formato_unix);
   printf("Hora do UNIX: %s\n", asctime(local));
 }
```

USANDO TIMEZONE PARA CALCULAR A DIFERENÇA ENTRE A ZONA HORÁRIA 637

Como foi visto, a biblioteca de execução de C fornece várias funções que podem converter valores de hora entre o horário local e o Horário de Greenwich. Para ajudar seus programas a determinar rapidamente a diferença de horas entre dois horários, muitos compiladores C oferecem a função *timezone*, que contém o número de segundos entre os dois horários. O programa a seguir, *timezone.c*, usa a variável global *timezone* para exibir a diferença de horário:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
  tzset();
  printf("Diferença entre a hora local e GMT é %d horas\n", timezone / 3600);
 }
```

Nota: A função tzset usa o item TZ no ambiente para determinar a zona horária correta.

638 DETERMINANDO A ZONA HORÁRIA ATUAL

Várias dicas nesta seção apresentaram funções que calculam horários com base na zona horária atual. Para ajudar os programas a determinar a zona horária atual, muitos compiladores fornecem a variável global *tzname*. A variável contém dois ponteiros: *tzname[0]* aponta para o nome da zona horária de três caracteres e *tzname[1]* aponta para o nome de três letras da zona de horário de verão. O programa a seguir, *tzname.c*, usa a variável global *tzname* para exibir os nomes das zonas horárias atuais:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   tzset();
   printf("A zona horária atual é %s\n", tzname[0]);
   if (tzname[1])
      printf("A zona de horário de verão é %s\n", tzname[1]);
   else
      printf("A zona de horário de verão não está definida\n");
 }
```

Nota: A função tzset usa a variável do ambiente TZ para determinar a zona horária atual.

639 DEFININDO CAMPOS DE ZONA HORÁRIA COM TZSET

Várias funções e variáveis globais nesta seção retornam informações sobre a zona horária atual. Muitas funções chamam a função *tzset* para determinar as informações de zona horária, como mostrado aqui:

```
#include <time.h>

void tzset(void);
```

A função *tzset* usa a variável de ambiente *TZ* para determinar as definições da zona horária atual. A função então atribui valores de propriedade para as variáveis globais *timezone*, *daylight* e *tzname*. O programa *tzname.c*, apresentado na dica anterior, ilustra como usar a função *tzset*.

640 USANDO A VARIÁVEL DE AMBIENTE TZ

Muitas dicas apresentadas nesta seção dependem da função *tzset* para fornecer informações de zona horária. A função *tzset* procura a variável *TZ* entre as suas variáveis do ambiente e depois atribui as variáveis *daylight*, *timezone* e *tzname*, com base no valor da variável. Você pode usar ou o comando SET do DOS para atribuir um valor para a variável TZ ou a definição Data/Hora dentro do Painel de Controle do Windows. Quando você usar o comando SET, o formato da entrada será:

```
TZ=SSS[+/-]h[h][DDD]
```

Onde *SSS* contém o nome da zona horária padrão (por exemplo, EST ou PST), o [+/-]h[h] especifica a diferença em horas entre a zona horária padrão e GMT; e *DDD* especifica o nome para a zona horária de verão (por exemplo, PDT). A entrada a seguir define a zona horária para a costa oeste norte-americana quando o horário de verão estiver ativo:

```
C:\> SET TZ=PST8PDT   <Enter>
```

Omita o nome da zona horária quando o horário de verão não estiver ativo, como mostrado aqui:

```
C:\:>  SET  TZ=PST8   <Enter>
```

Experimente a variável de ambiente TZ e os programas de zona horária apresentadas nesta seção para determinar como você quer representar datas e horas dentro dos seus programas. Lembre-se, no entanto, de que seus programas precisarão escrever a entrada TZ em qualquer computador para o qual você mover seus aplicativos.

Nota: Se você não especificar a variável TZ, o padrão será EST5EDT.

DEFININDO A ENTRADA DE AMBIENTE TZ A PARTIR DE DENTRO DE SEU PROGRAMA 641

Como você aprendeu, várias funções da biblioteca de execução de C usam *tzset* para determinar a zona de tempo local. Como discutido na dica anterior, a função *tzset* usa a entrada de ambiente TZ para determinar a zona horária. Na maioria dos casos, provavelmente não é razoável esperar que os usuários finais definam corretamente a entrada do ambiente TZ. No entanto, se você souber a definição correta para um usuário específico, poderá usar a função *putenv* dentro do programa para criar a entrada correta para esse usuário, como mostrado aqui:

```
putenv("TZ=PST8PDT");
```

O programa a seguir, *def_tz.c*, usa a função *putenv* para definir a zona horária correta. O programa então usa a variável global *tzname* para exibir as definições da zona horária, como mostrado aqui:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void main(void)
 {
   putenv("TZ=PST8PDT");
   tzset();
   printf("A zona horária atual é %s\n", tzname[0]);
   if (tzname[1])
     printf("A zona de horário de verão é %s\n",tzname[1]);
   else
     printf("A zona de horário de verão não está definida\n");
 }
```

OBTENDO AS INFORMAÇÕES DE ZONA HORÁRIA 642

Várias dicas nesta seção apresentam modos de seus programas determinarem as informações de zona horária. Uma das funções mais úteis que seus programas podem usar para obter as informações de zona horária é *ftime*, como mostrado aqui:

```
#include <sys\timeb.h>

void ftime(struct timeb *zonahora);
```

O parâmetro *zonahora* é um ponteiro para uma estrutura do tipo *timeb*, como mostrado aqui:

```
struct timeb
 {
   long time;
   short millitm;
   short timezone;
   short dstflag;
 };
```

O campo *time* contém o número de segundos desde 1/1/1970 (GMT). O campo *millitm* contém a parte fracionária dos segundos em milissegundos. O campo *timezone* contém a diferença entre a zona horária local e GMT em minutos. Finalmente, o parâmetro *dstflags* especifica se o horário de verão está ativo (se o valor do sinalizador é 1) ou inativo (se o valor do sinalizador é 0). O programa a seguir, *ftime.c*, usa a função *ftime* para exibir as informações atuais de zona horária:

```
#include <stdio.h>
#include <time.h>
#include <sys\timeb.h>

void main(void)
```

```
{
 struct timeb zonahora;
 tzset();
 ftime(&zonahora);
 printf("Segundos desde 1/1/1970 (GMT) %ld\n", zonahora.time);
 printf("Segundos fracionários %d\n", zonahora.millitm);
 printf("Diferença de horas entre GMT e zona local %d\n",
    zonahora.timezone / 60);
 if (zonahora.dstflag)
    printf("Horário de verão ativo\n");
 else
    printf("Horário de verão inativo\n");
}
```

643 DEFININDO A HORA DO SISTEMA EM SEGUNDOS DESDE A MEIA-NOITE DE 1/1/1970

Várias Dicas nesta seção mostram modos de definir o temporizador do sistema usando o DOS e a BIOS. Além dos métodos discutidos anteriormente, seus programas também podem usar a função *stime* para definir o horário do sistema usando segundos desde a meia-noite de 1/1/1970, como mostrado aqui:

```
#include <time.h>

int stime(time_t *segundos);
```

A função *stime* sempre retorna 0. O programa a seguir, *stime.c*, usa a função *stime* para definir a data exatamente um dia após a data e a hora atuais:

```
#include <time.h>

void main(void)
 {
   time_t segundos;

   time(&segundos);    // Obtém o horário atual
   segundos += (time_t) 60 * 60 * 24;
   stime(&segundos);
 }
```

644 CONVERTENDO UMA DATA PARA SEGUNDOS DESDE A MEIA-NOITE DE 1/1/1970

Várias dicas neste livro apresentam funções da biblioteca de execução que usam ou retornam os segundos desde a meia noite de 1/1/1970. Para lhe ajudar a determinar os segundos para uma data específica, seus programas podem usar a função *mktime*, como mostrado aqui:

```
#include <time.h>

time_t mktime(struct tm *campos_hora);
```

Se os campos de hora são válidos, a função retorna o número de segundos para a hora especificada. Se um erro ocorrer, a função retornará -1. O parâmetro *campos_hora* é um parâmetro para uma estrutura do tipo *tm*, como mostrado aqui:

```
struct tm
 {
   int tm_sec;
   int tm_min;
   int tm_hour;
```

```
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

O programa a seguir, *mktime.c*, usa a função *mktime* para determinar o número de segundos entre a meia-noite de 1/1/1970 e a meia-noite de 31/10/1997:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   time_t segundos;

   struct tm campos_hora;
   campos_hora.tm_mday = 22;
   campos_hora.tm_mon = 8;
   campos_hora.tm_year = 97;
   campos_hora.tm_hour = 0;
   campos_hora.tm_min = 0;
   campos_hora.tm_sec = 0;
   segundos = mktime(&campos_hora);
   printf("O número de segundos entre 1/1/70 e 31/10/97 é %ld\n", segundos);
 }
```

Nota: Quando você passa uma estrutura tm parcial para a função mktime, a função preenche os campos que não são corretos. A função mktime suporta datas no intervalo de 1/1/1970 até 19/1/2028.

DETERMINANDO A DATA JULIANA DE UMA DATA 645

Na dica anterior você usou a função *mktime* para determinar o número de segundos entre uma data específica e a meia-noite de 1/1/1970. Como você aprendeu, a função *mktime* usa uma estrutura do tipo *tm* para armazenar os componentes da data. Se um ou mais dos componentes não estão completos, a função *mktime* preenche-os. Se você examinar a estrutura *tm*, verá o membro *tm_yday*. Quando você chama *mktime*, a função atribuirá a *tm_yday* a data juliana para o dia especificado. O calendário juliano é idêntico ao gregoriano, exceto que inicia o ano 1 no ano gregoriano equivalente a 46 a.C. O computador informa as datas julianas em um formato de três dígitos. O programa a seguir, *juliana.c*, usa a função *mktime* para determinar a data juliana para 31/10/1997.

```
#include <stdio.h>

#include <time.h>

void main(void)
 {
   time_t segundos;
   struct tm campos_hora;

   campos_hora.tm_mday = 31;
   campos_hora.tm_mon = 10;
   campos_hora.tm_year = 97;
   if (mktime(&campos_hora) == -1)
     printf("Erro ao converter os campos\n");
   else
     printf("A data juliana para 31/10/1997 é %d\n", campos_hora.tm_yday);
 }
```

646 CRIANDO UMA STRING FORMATADA DE DATA E HORA

Como você aprendeu, as funções *_strdate* e *_strtime* retornam a data e a hora atuais em um formato de string de caracteres. Muitos compiladores oferecem a seguinte função *strftime* para que você tenha um melhor controle de uma string formatada de data e hora, como mostrado aqui:

```
#include <time.h>

size_t strftime(char *string, size_t tamanho_max,
        const char *formato, const struct tm *datahora);
```

O parâmetro *string* é a string de caracteres na qual *strftime* escreve a string de data e hora formatada. O parâmetro *tamanho_max* especifica o número máximo de caracteres que *strftime* pode colocar na string. A string *formato* usa caracteres de formatação *%letra* similares a *printf* para especificar o formato que você deseja. A Tabela 646 lista os caracteres válidos que você pode colocar na string formatada. Finalmente, o parâmetro *datahora* é um ponteiro para uma estrutura do tipo *tm* que contém os campos de data e hora. A função *strftime* retorna um contador do número de caracteres atribuído ao parâmetro *string*, ou 1 se a função extravasou o parâmetro *string*. A Tabela 646 lista os especificadores de formato para *strftime*.

Tabela 646 Especificadores de formato para a função ***strftime***.

Especificador de Formato	Significado
%%	Caractere %
%a	Nome abreviado do dia da semana
%A	Nome completo do dia da semana
%b	Nome do mês abreviado
%B	Nome do mês completo
%c	Data e hora
%d	Dia do mês em dois dígitos, de 01 a 31
%H	Hora em dois dígitos, de 00 a 23
%I	Hora em dois dígitos, de 01 a 12
%j	Dia juliano em 3 dígitos
%m	Mês decimal de 1 a 12
%M	Minutos em dois dígitos, de 00 a 59
%p	Caracteres AM ou PM
%S	Segundos de dois dígitos de 00 a 59
%U	Número da semana em dois dígitos, de 00 até 53, sendo domingo o primeiro dia da semana
%w	Dia da semana (0 = domingo, 6 = sábado)
%W	Número da semana em dois dígitos, de 00 até 53, sendo segunda o primeiro dia da semana
%x	Data
%X	Hora
%y	Ano em dois dígitos, de 00 a 99
%Y	Ano em quatro dígitos
%Z	Nome da zona horária

O programa a seguir, *strftime.c*, ilustra a função *strftime*:

```
#include <stdio.h>
#include <time.h>

void main(void)
 {
   char buffer[128];
   struct tm *datahora;
   time_t hora_atual;

   tzset();
```

```
    time(&hora_atual);
    datahora = localtime(&hora_atual);
    strftime(buffer, sizeof(buffer), "%x %X", datahora);
    printf("Usando %%x %%X: %s\n", buffer);
    strftime(buffer, sizeof(buffer), "%A %B %d, %Y", datahora);
    printf("Usando %%A %%B %%d %%Y: %s\n", buffer);
    strftime(buffer, sizeof(buffer), "%I:%M%p", datahora);
    printf("Usando %%I:%%M%%p: %s\n", buffer);
}
```

Quando você compilar e executar o programa *strftime.c*, sua tela exibirá saída similar ao seguinte (sua tela exibirá saída com base na data e hora atuais):

```
Usando %x %X: 22/08/97 22:03:13
Usando %A %B %m %Y: Friday August 22, 1997
Usando %I:%M%p: 10:03PM
C:\>
```

COMPREENDENDO OS TIPOS DE RELÓGIO DO PC 647

Várias dicas nesta seção discutem datas e horas do PC. Para compreender melhor essas funções, você precisa saber que o PC usa quatro tipos básicos de relógios — temporizadores, o relógio da CPU, o relógio de tempo real e o relógio da CMOS — detalhados na lista a seguir:

- O relógio do temporizador é um circuito integrado dentro do PC que gera uma interrupção 18,2 vezes por segundo. Cada vez que ocorre um pulso do relógio, o PC gera a interrupção 8 (uma mensagem do sistema). Capturando essa interrupção, os programas residentes na memória podem ativar a si mesmos em intervalos de tempos específicos.
- O relógio da CPU controla com que velocidade seus programas são executados. Quando os usuários dizem que estão usando um sistema de 200MHz, estão referenciando o relógio da CPU.
- O relógio de tempo real controla a data e a hora atuais. Em muitos casos, o relógio de tempo real contém o mesmo valor que o relógio da CMOS.
- O relógio da CMOS é mantido pelo computador, ao contrário do sistema operacional, que mantém o relógio de tempo real. O relógio da CMOS geralmente contém a mesma entrada que o relógio de tempo real.

AGUARDANDO A DIGITAÇÃO DE UMA TECLA 648

Existem muitos programas que, ao exibir uma mensagem, irão esperar que o usuário pressione uma tecla antes de remover a mensagem e continuar. Para ajudar seus programas a efetuar esse processamento, você poderá usar a seguinte função *kbhit*, que retornará o valor verdadeiro se o usuário pressionar uma tecla, e falso se o usuário não pressionar uma tecla:

```
#include <conio.h>

int kbhit(void);
```

O programa a seguir, *kbhit.c*, exibirá uma mensagem na tela que pede que o usuário pressione qualquer tecla para continuar. O programa então usa *kbhit* para aguardar a digitação, como mostrado aqui:

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
  printf("Pressione qualquer tecla para continuar...");
  while (! kbhit())
    ;
  printf("Acabado\n");
}
```

649 PEDINDO UMA SENHA AO USUÁRIO

Dependendo dos seus programas, algumas vezes você precisa pedir uma senha ao usuário. Quando o usuário digitar a senha, as teclas que ele digitar não deverão aparecer na tela. Seus programas podem usar a seguinte função *getpass* para executar esta tarefa:

```
#include <conio.h>

char *getpass(const char *prompt);
```

A função *getpass* exibirá a mensagem especificada, e depois, esperará que o usuário digite teclas e pressione Enter. A função *getpass* então retorna um ponteiro para a senha que o usuário digitou. O programa a seguir, *getpass.c*, usa a função *getpass* para pedir que o usuário digite uma senha:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

void main(void)
 {
   char *senha;

   senha = getpass("Informe a senha:");
   if (strcmp(senha, "Biblia"))
     printf("Senha incorreta\n");
   else
     printf("Senha OK\n");
 }
```

Nota: Caso seu compilador não forneça uma função getpass, você pode usar a função le_senha, mostrada na Dica 650.

650 ESCREVENDO SUA PRÓPRIA FUNÇÃO DE SENHA

Na dica anterior você aprendeu como usar a função *getpass* para pedir uma senha ao usuário. Como você sabe, *getpass* não exibe as teclas digitadas pelo usuário. Alguns novos usuários terão dificuldade em digitar uma senha se a tela não mostrar nada, de modo que alguns programas exibem um asterisco à medida que o usuário vai digitando. Para pedir que o usuário digite uma senha e exibir um asterisco para cada tecla digitada, você pode usar a função *le_senha*, como mostrado aqui, dentro do programa *digi_sen.c*:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>

#define BACKSPACE 8
char *le_senha(const char *prompt)
 {
   static char buffer[128];
   int i = 0;
   char letra = NULL;

   printf(prompt);
   while ((i < 127) && (letra != '\r'))
     {
       letra = getch();
       if (letra == BACKSPACE)
         {
           if (i > 0)
            {
              buffer[--i] = NULL;  // Apaga o * anterior
              putchar(BACKSPACE);
              putchar(' ');
```

```
            putchar(BACKSPACE);
          }
        else
          putchar(7);  // Aviso sonoro
      }
    else if (letra != '\r')
      {
        buffer[i++] = letra;
        putchar('*');
      }
  }
  buffer[i] = NULL;
  return (buffer);
}

void main(void)
{
  char *senha;

  senha = le_senha("Informe a senha: ");
  if (strcmp(senha, "Biblia"))
    printf("\nSenha incorreta\n");
  else
    printf("\nSenha OK\n");
}
```

COMPREENDENDO O REDIRECIONAMENTO DA SAÍDA 651

Cada vez que você executa um comando, o sistema operacional associa o dispositivo de entrada padrão ao seu teclado. O sistema operacional referencia o monitor como o dispositivo de saída padrão, ou *stdout*. Usando o operador de redirecionamento da saída (>), você pode instruir o sistema operacional a rotear a saída de um programa a um arquivo ou a algum outro dispositivo. O comando a seguir, por exemplo, instrui o DOS a redirecionar a saída do comando *dir* da tela para a impressora:

```
C:\> DIR  > PRN   <Enter>
```

De um modo similar, o comando a seguir instrui o DOS a redirecionar a saída do comando *chkdsk* para o arquivo *infodisc.dat*:

```
C:\>  CHKDSK  >  INFODISC.DAT    <Enter>
```

Para lhe ajudar a escrever programas que suportam o redirecionamento da saída, o arquivo de cabeçalho *stdio.h* define a constante *stdout* para a qual as operações de saída de arquivo podem direcionar a saída. Várias dicas apresentadas neste livro escrevem saída para *stdout*.

COMPREENDENDO O REDIRECIONAMENTO DA ENTRADA 652

Sempre que você executa um comando, o sistema operacional associa o dispositivo de saída padrão à tela do computador. O sistema operacional referencia o teclado como o dispositivo de entrada padrão, ou *stdin*. Você pode usar o operador de redirecionamento da entrada (<) para direcionar o sistema operacional a rotear a entrada de um programa do teclado para um arquivo ou para algum outro dispositivo. Por exemplo, o comando a seguir instrui o DOS a redirecionar a entrada para o comando *more* do teclado para o arquivo *config.sys*:

```
C:\>  MORE  < CONFIG.SYS   <Enter>
```

De um modo similar, o comando a seguir instrui o DOS a redirecionar a entrada do comando *sort* do teclado para o arquivo *autoexec.bat*:

```
C:\> SORT  < AUTOEXEC.BAT   <Enter>
```

Para lhe ajudar a escrever programas que suportam o redirecionamento da entrada, o arquivo de cabeçalho *stdio.h* define as constantes *stdin*, a partir da qual as operações de entrada podem obter entrada. Várias dicas apresentadas neste livro lêem entrada de *stdin*.

653 COMBINANDO O REDIRECIONAMENTO DA ENTRADA E DA SAÍDA

Como discutido nas Dicas 651 e 652, você pode alterar a fonte de entrada e saída padrão de um programa do teclado e monitor usando os operadores de entrada (<) e saída (>). À medida que você criar uma coleção de programas que suporta o redirecionamento da entrada e saída, algumas vezes você desejará redirecionar as fontes de entrada e saída de um programa no mesmo comando. Por exemplo, o comando a seguir instrui o DOS a classificar o conteúdo do arquivo *config.sys* e enviar a saída classificada para a impressora:

```
C:\> SORT < CONFIG.SYS > PRN <Enter>
```

Para compreender o processamento que o sistema operacional executa, leia o comando da esquerda para a direita. O operador de redirecionamento da entrada (<) instrui *sort* a obter sua entrada a partir do arquivo *config.sys*. Da mesma forma, o operador de redirecionamento da saída (>) instrui a saída de *sort* do monitor para a impressora.

654 USANDO STDOUT E STDIN

Nas Dicas 651 e 652, você aprendeu que C define os indicativos de arquivo *stdin* e *stdout*. Os indicativos de arquivo lhe permitem escrever programas que suportam o redirecionamento da E/S. O programa a seguir, *maiusc.c*, lê uma linha de texto do indicativo de arquivo *stdin* e converte o texto para maiúsculas. O programa então escreve a linha de texto em *stdout*. O programa continua a escrever texto até detectar o final do arquivo:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char linha[255];  // Linha de texto lida

   while (fgets(linha, sizeof(linha), stdin))
     fputs(strupr(linha), stdout);
 }
```

Usando o comando *maiusc*, você pode exibir o conteúdo do arquivo *autoexec.bat*, como mostrado aqui:

```
C:\> MAIUSC < AUTOEXEC.BAT <Enter>
```

O comando a seguir usa o operador de redirecionamento para imprimir o conteúdo do arquivo *config.sys* em maiúsculas:

```
C:\> MAIUSC < CONFIG.SYS > PRN <Enter>
```

Se você chamar o comando *maiusc* sem usar o operador de redirecionamento de E/S, como mostrado no exemplo a seguir, o comando *maiusc* lerá sua entrada a partir do teclado e escreverá sua saída na tela:

```
C:\> MAIUSC <Enter>
```

Toda vez que você digitar uma linha de texto e pressionar Enter, *maiusc* exibirá o texto correspondente em maiúsculas. Para finalizar o programa, você precisa pressionar a combinação de teclas de fim de arquivo Ctrl+Z (sob o DOS) ou Ctrl+D (sob o UNIX).

655 COMPREENDENDO O OPERADOR DE CANALIZAÇÃO

Nas Dicas 651 e 652, você aprendeu como usar os operadores de redirecionamento de entrada e saída para alterar a origem de entrada de um programa do teclado para um arquivo ou dispositivo. Você também aprendeu

que pode usar operadores de redirecionamento de entrada e saída para rotear a saída de um programa da tela para um arquivo ou dispositivo. O DOS e o UNIX também fornecem um terceiro operador de redirecionamento, chamado *operador de canalização*, o que lhe permite redirecionar a saída de um programa a se tornar a entrada de outro programa. Por exemplo, o comando a seguir instrui o DOS a redirecionar a saída do comando *dir* para se tornar a entrada do comando *sort*:

```
C:\> DIR | SORT   <Enter>
```

Os programas que recebem sua entrada a partir de outro comando ou arquivo e depois mudam a entrada de alguma forma são chamados *filtros*. Por exemplo, o comando a seguir usa o comando *find* para filtrar a saída do comando *dir* para exibir apenas as entradas de subdiretório:

```
C:\> DIR | FIND "<DIR>"   <Enter>
```

Exatamente como você pode usar múltiplos operadores de redirecionamento de entrada e saída na mesma linha de comando, também pode colocar dois ou mais comandos de canalização para exibir os nomes dos subdiretórios na ordem classificada, uma tela de cada vez:

```
C:\> DIR | FIND "<DIR>" | SORT | MORE   <Enter>
```

COMPREENDENDO GETCHAR E PUTCHAR 656

Muitos programas usam as macros *getchar* e *putchar* para entrada e saída de caracteres. Por exemplo, o programa a seguir, *minusc.c*, converterá cada linha que o usuário digitar para minúsculas e depois exibirá cada linha de texto digitada na tela:

```
#include <stdio.h>
#include <ctype.h>  // Contém o protótipo de tolower

void main(void)
 {
    int letra;

    for (letra = getchar(); ! feof(stdin); letra = getchar())
      putchar(tolower(letra));
 }
```

O comando a seguir usa o programa *minusc* para imprimir o conteúdo do arquivo *autoexec.bat* para minúsculas:

```
C:\> MINUSC < AUTOEXEC.BAT > PRN <Enter>
```

Quando você usa as macros *getchar* e *putchar*, seus programas automaticamente suportam o redirecionamento da E/S. Para compreender melhor como ocorre o redirecionamento da E/S, examine o arquivo de cabeçalho *stdio.h*. Dentro de *stdio.h*, você encontrará as macros *getchar* e *putchar*, que definem suas fontes de entrada e saída em termos de *stdin* e *stdout*, como mostrado aqui:

```
#define getchar()   getc(stdin)
#define putchar(c)  putc((c), stdout)
```

NUMERANDO A ENTRADA REDIRECIONADA 657

Dependendo do conteúdo de um arquivo ou a saída de um programa, algumas vezes você poderá querer preceder cada linha do conteúdo do arquivo ou saída do programa com um número de linha. O programa a seguir, *numero.c*, filtra sua entrada para preceder cada linha com o número de linha correspondente da linha:

```
#include <stdio.h>

void main(void)
 {
   char linha[255];  // Linha de entrada
```

```
    long num_linha = 0;  // Número da linha atual

    while (fgets(linha, sizeof(linha), stdin))
       printf("%ld %s", ++num_linha, linha);
}
```

Por exemplo, o comando a seguir imprime uma cópia do arquivo *numero.c*, com um número de linha precedendo cada linha:

```
C:\> NUMERO < NUMERO.C > PRN <Enter>
```

658 GARANTINDO QUE UMA MENSAGEM APAREÇA NA TELA

Você pode usar os operadores de redirecionamento de saída e de canalização para redirecionar a saída de um programa da tela para um arquivo, para um dispositivo ou para a entrada de outro programa. Embora esse redirecionamento da saída possa vir a ser uma ferramenta poderosa, ele também pode fazer os usuários perderem uma mensagem de erro se não observarem atentamente seu trabalho. Para compreender melhor, considere o programa a seguir, *novotipo.c*, que exibe o conteúdo de um arquivo na tela:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   char linha[255];  // Linha lida do arquivo
   FILE *pa;

   if (pa = fopen(argv[1], "r"))
     {
       while (fgets(linha, sizeof(linha), pa))
          fputs(linha, stdout);
       fclose(pa);
       exit(0);  // Bem-sucedida
     }
   else
     {
       printf("Não foi possível abrir %s\n", argv[1]);
       exit (1);
     }
 }
```

No entanto, se você quisesse enviar a saída de um programa para um dispositivo ou arquivo, poderia usar o operador de redirecionamento da saída para redirecionar a saída do programa. O comando a seguir, por exemplo, redireciona a saída de *novotipo* para imprimir o arquivo *autoexec.bat*:

```
C:\> NOVOTIPO AUTOEXEC.BAT > PRN    <Enter>
```

Se *novotipo* abrir com sucesso *autoexec.bat*, ele escreverá o conteúdo do arquivo em *stdout*, que, baseado no redirecionamento da saída, fará o arquivo ser impresso. Se o programa *novotipo* não puder abrir o arquivo que você especificar, ele usará a função *printf* para exibir uma mensagem de erro dizendo que não foi possível abrir o arquivo. Infelizmente, devido ao redirecionamento da saída, a mensagem não aparece na tela; em vez disso, ela vai para a impressora. Os usuários poderiam erroneamente acreditar que o comando foi bem-sucedido a não ser que, de imediato, verificassem a impressão. Para impedir que seu programa inadvertidamente redirecione suas mensagens de erro, C define o indicativo de arquivo *stderr*, que seus programas não podem redirecionar a partir da tela. Quando seus programas precisarem exibir uma mensagem de erro na tela, deverão usar *fprintf* para escrever a mensagem em *stderr*, como mostrado aqui:

```
fprintf(stderr, "Não foi possível abrir %s\n", argv[1]);
```

ESCREVENDO SEU PRÓPRIO COMANDO MORE 659

Um dos melhores filtros que o DOS e o UNIX fornecem é o comando *more*, que exibe sua entrada uma tela por vez. Cada vez que *more* exibe uma tela de saída, *more* faz uma pausa, aguarda que o usuário pressione uma tecla e exibe a seguinte mensagem:

```
— More —
```

Quando o usuário pressiona uma tecla, *more* repete o processo e exibe a próxima tela de saída. O programa a seguir, *more.c*, implementa o comando *more:*

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   char buffer[256];
   long conta_linha = 0;
   union REGS regs_ent, regs_sai;
   int Ctrl_tecla_press, cdgvarredura;

   while (fgets (buffer, sizeof(buffer), stdin))
   {
      fputs (buffer, stdout);
      if ((++conta_linha % 24) == 0)
      {
         printf ("-- Mais --");
         // Pega o cdg de varredura da tecla
         regs ent.h.ah = 0;
         int86 (0x16, &regs_ent, &regs_sai);
         cdgvarredura = regs sai.h.ah;
         // Pega o estado do teclado no caso de Ctrl-C
         Ctrl_tecla_press = 0;
         regs_ent.h.ah = 2;
            int86 (0x16, &regs_ent, &regs_sai);
         // O sinalizador da tecla Ctrl é o bit 2
         Ctrl_tecla_press = (regs_sai.h.al & 4);
         // cdgvarredura para C é 0x2E
         if ((ctrl_tecla_press) && (cdgvarredura==0x2E))
             break;      // Ctrl-C pressionada
          printf ("\r");
        }
    }
 }
```

Cada vez que *more* faz uma pausa para o usuário pressionar uma tecla, ele chama a interrupção de teclado da BIOS (INT 16H) para receber a digitação. Como o DOS define suas operações de entrada em termos de *stdin*, você não pode usar *getchar*, *getch* ou *kbhit* para ler a digitação. As funções de entrada do DOS usam a próxima entrada redirecionada, desse modo tratando o próximo caractere redirecionado como a digitação do usuário. No entanto, os serviços da BIOS não são definidos em termos de *stdin*, e, portanto, o operador de redirecionamento não afeta os serviços de entrada da BIOS.

EXIBINDO UM CONTADOR DE LINHAS REDIRECIONADAS 660

Várias dicas nesta seção criaram comandos de filtros que você pode usar com os operadores de redirecionamento de *entrada* e *canalização*. O programa a seguir, *contalin.c*, exibirá um contador do número de linhas de entrada redirecionadas:

```
#include <stdio.h>

void main(void)
 {
   char linha[256];  // Linha de entrada redirecionada
   long conta_linha = 0;

   while (fgets(linha, sizeof(linha), stdin))
      conta_linha++;
   printf("Número de linhas redirecionadas: %ld\n", conta_linha);
 }
```

661 EXIBINDO UM CONTADOR DE CARACTERES REDIRECIONADOS

Muitas dicas nesta seção criaram comandos de filtro que você pode usar com os operadores de redirecionamento de entrada e de canalização do DOS. De um modo similar, o programa a seguir, *contacar.c*, exibirá o número de caracteres na entrada redirecionada:

```
#include <stdio.h>

void main(void)
 {
   long conta_carac = 0;
   getchar();
   while (! feof(stdin))
     {
        getchar();
        conta_carac++;
     }
   printf("Número de caracteres redirecionados: %ld\n", conta_carac);
 }
```

662 CRIANDO UM COMANDO MORE COM CONTROLE DE TEMPO

Várias dicas nesta seção criaram comandos de filtro que você pode usar com os operadores de redirecionamento de *entrada* e de *canalização* do Dos. De um modo similar, o programa a seguir, *more15.c*, muda o comando *more* do DOS para exibir uma tela de entrada redirecionada com cada digitação a cada 15 segundos, o que vier primeiro:

```
#include <stdio.h>
#include <time.h>
#include <dos.h>

void main(void)
 {
   char buffer[256];
   char tecla_press = 0;
   long int contador = 1;
   union REGS regs_ent, regs_sai;

   time_t hora_inicio, hora_atual, hora_final;
   while (fgets(buffer, sizeof(buffer), stdin))
     {
       fputs (buffer, stdout);
       if ((++contador % 25) == 0)
       {
         time (&hora_inicio);
         hora_final = hora_inicio + 15;
         do
```

```
        {
          tecla_press = 0;
          time (&hora_atual);
          regs_ent.h.ah = 1;
          int86 (0x16, &regs_ent, &regs_sai);
          if ((regs_sai.x.flags & 64) == 0)
          {
            tecla_press = 1;
            do
              {
                regs_ent.h.ah = 0;
                int86 (0x16, &regs_ent, &regs_sai);
                regs_ent.h.ah = 1;
                int86 (0x16, &regs_ent, &regs_sai);
              } while (! (regs_sai.x.flags & 64));
          }
        }
      while ((hora_atual != hora_final) &&
        (! tecla_press));
    }
  }
}
```

IMPEDINDO O REDIRECIONAMENTO DA E/S 663

Como foi visto, ao criar programas que suportam o redirecionamento da E/S, você pode criar uma biblioteca de comandos de filtro poderosos. No entanto, muitos programas que você criará não suportam o redirecionamento da E/S. Dependendo das funções que seu programa executa, erros severos podem ser o resultado quando você permite o redirecionamento O programa a seguir, *semredir.c*, testa os indicativos de arquivo *stdin* e *stdout* para garantir que eles não tenham sido redirecionados.

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   union REGS regs_ent, regs_sai;

   // Checa o indicativo stdin primeiro
   regs_ent.x.ax = 0x4400;
   regs_ent.x.bx = 0;       // stdin é o indicativo 0
   intdos (&regs_ent, &regs_sai);
   if ((regs_sai.x.dx & 1) && (regs_sai.x.dx & 128))
     fprintf (stderr, "stdin não foi redirecionado\n");
   else
     fprintf (stderr, "stdin está redirecionado\n");
   // Agora checa stdout
   regs_ent.x.ax = 0x4400;
   regs_ent.x.bx = 1;       // stdout é o indicativo 1
   intdos (&regs_ent, &regs_sai);
   if ((regs_sai.x.dx & 2) && (regs_sai.x.dx & 128))
     fprintf (stderr, "stdout não foi redirecionado\n");
   else
     fprintf (stderr, "stdout está redirecionado\n");
 }
```

O programa usa o serviço 4400H da INT 21H do DOS para examinar o indicativo de arquivo. Se o indicativo aponta para um dispositivo, então o serviço liga o bit 7 do registrador DX. Se o serviço liga o bit 7 e o bit 2, então o indicativo se refere a *stdout*. Se o serviço liga o bit 7 e o bit 1, então o indicativo se refere a *stdin*.

Se o serviço não liga o bit 7, então o programa redirecionou o indicativo para um arquivo. Se o serviço nao liga o bit 1 ou o bit 2, então o programa redirecionou o indicativo para um dispositivo diferente de *stdin* ou *stdout*.

Seus programas podem usar o serviço *4400H* da *INT 21H* para determinar se o programa atual, um programa executado anteriormente, ou se o usuário redirecionou a entrada ou a saída do computador. Dependendo do resultado do teste, os programas podem agir de forma apropriada.

664 USANDO O INDICATIVO DE ARQUIVO STDPRN

Como você aprendeu, o arquivo de cabeçalho *stdio.h* define dois indicativos de arquivo — *stdin*, que (por padrão) aponta para o teclado, e *stdout*, que aponta para a tela. Se você escrever operações de entrada e de saída em termos de *stdin* e *stdout*, seus programas automaticamente suportarão o redirecionamento da E/S. De um modo similar, *stdio.h* define o indicativo de arquivo *stdprn*, que aponta para o dispositivo de impressão padrão (PRN ou LPT1). Ao contrário de *stdin* e *stdout*, você não pode redirecionar *stdprn*. O programa a seguir, *eco_ptr.c*, usa o arquivo *stdprn* para imprimir entrada redirecionada à medida que o programa exibir a saída na tela usando *stdout*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char linha[255];  // Linha de texto lida

   while (fgets(linha, sizeof(linha), stdin))
     {
       fputs(linha, stdout);
       strcat(linha, "\r");
       fputs(linha, stdprn);
     }
 }
```

A linha de comando a seguir usa o programa *eco_ptr* para imprimir e exibir uma listagem de diretório:

```
C:\> DIR  |  ECO_PTR   <Enter>
```

665 DESVIANDO A SAÍDA REDIRECIONADA PARA UM ARQUIVO

Quando você usa o operador de *canalização* do DOS para redirecionar a saída de um programa para se tornar a entrada de outro programa, algumas vezes você pode querer salvar uma cópia intermediária da saída de um programa para um arquivo. O programa a seguir, *tee.c*, salva uma cópia intermediária da saída de um programa em um arquivo:

```
#include <stdio.h>

void main(void)
 {
   char buffer[256];

   while (fgets(buffer, sizeof(buffer), stdin))
     {
       fputs(buffer, stdout);
       fputs(buffer, stderr);
     }
 }
```

O comando *tee* escreve sua entrada redirecionada para um arquivo que você especifica, e para *stdout*, para que seu programa possa redirecionar sua saída para outro programa. O comando a seguir, por exemplo, usa *tee* para imprimir uma listagem de diretório não-classificado antes de o comando *sort* exibir a lista de diretório classificada na sua tela:

```
C:\> DIR  |  TEE  PRN  |  SORT    <Enter>
```

USANDO O INDICATIVO DE ARQUIVO STDAUX 666

Como você aprendeu, o arquivo de cabeçalho *stdio.h* define três indicativos de arquivo — *stdin*, que (por padrão) aponta para o teclado; *stdout*, que (por padrão) aponta para a tela; e *stdprn*, que sempre aponta para a impressora. Se você escrever suas operações de entrada e saída em termos de *stdin* e *stdout*, seus programas automaticamente suportarão o redirecionamento da E/S. De um modo similar, *stdio h* define o indicativo de arquivo *stdaux*, que aponta para o dispositivo auxiliar padrão (AUX ou COM1). Ao contrário de *stdin* e *stdout*, você não pode redirecionar *stdaux*. O programa a seguir, *eco_aux.c*, usa o arquivo *stdaux* para enviar entrada redirecionada para COM1 à medida que o programa exibe saída para a tela usando *stdout*:

```
#include <stdio.h>
#include <string.h>

void main(void)
 {
   char linha[255];  // Linha de textp lido

   while (fgets(linha, sizeof(linha), stdin))
    {
       fputs(linha, stdout);
       strcat(linha, "\r");
       fputs(linha, stdaux);
    }
 }
```

A linha de comando a seguir usa *eco_aux.c* para imprimir (em uma impressora conectada a COM1) e exibir uma listagem de diretório:

```
C:\>  DIR  |  ECO_AUX    <Enter>
```

ENCONTRANDO OCORRÊNCIAS DE SUBSTRING DENTRO DA ENTRADA REDIRECIONADA 667

Muitas dicas apresentadas nesta seção criaram comandos de filtro que você poderá usar com os operadores de redirecionamento de *entrada* e de *canalização*. O programa a seguir, *local_es.c*, exibirá cada ocorrência de uma palavra ou frase dentro de sua entrada redirecionada:

```
#include <stdio.h>
#include <string.h>

void main(int argc, char *argv[])
 {
   char string[256];

   while (fgets(string, sizeof(string), stdin))
     if (strstr(string, argv[1]))
       fputs(string, stdout);
 }
```

Para exibir cada ocorrência da palavra *#include* dentro do arquivo *teste.c*, você pode chamar *local_es*, como segue:

```
C:\> LOCAL_ES  #include   <  TESTE.C   <Enter>
```

Para pesquisar duas ou mais palavras, simplesmente coloque as palavras dentro de aspas, como mostrado aqui:

```
C:\> LOCAL_ES "Os membros da" < ESTATUTO.DAT <Enter>
```

668 EXIBINDO AS PRIMEIRAS N LINHAS DA ENTRADA REDIRECIONADA

Várias dicas nesta seção criaram comandos de filtros que você poderá usar com os operadores de redirecionamento de *entrada* e de *canalização*. O programa a seguir, *exibprim.c*, exibe o número de linha que você especifica dentro da linha de comando da entrada redirecionada. Por padrão, o programa exibirá as primeiras 10 linhas de entrada redirecionada, como mostrado aqui:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   char linha[255];  // Linha lida do arquivo
   int i,j;
   FILE *pa;

   if (argc > 2)
      j = 10;
   else
      j = argv[2];
   if (pa = fopen(argv[1], "r"))
     {
       for (i=0; i < j; i++)
         {
            fgets(pa, sizeof(linha), linha);
            fputs(pa, stdout);
         }
       fclose(pa);
     }
   else
     {
       printf("Não pude abrir %s\n", argv[1]);
       exit (1);
     }
 }
```

Por exemplo, o comando a seguir instrui *exibprim* para exibir as primeiras 10 linhas de uma listagem de diretório redirecionada:

```
C:\> DIR | EXIBPRIM <Enter>
```

O comando a seguir, por outro lado, instrui *exibprim* a exibir as primeiras 25 linhas de uma listagem de diretório redirecionada.

```
C:\> DIR | EXIBPRIM 25 <Enter>
```

669 COMPREENDENDO OS ARGUMENTOS DA LINHA DE COMANDO

Quando você executa comandos, os caracteres que digita após o comando e antes de pressionar a tecla Enter são a linha de comando do programa. Por exemplo, o comando a seguir chama o programa *primeiro* usando dois argumentos, o número de linhas a exibir e o nome de arquivo que o usuário deseja:

```
C:\> PRIMEIRO 15 NOMEARQ.EXT <Enter>
```

O suporte para os argumentos da linha de comando aumenta o número de aplicações nas quais você pode usar seu programa. Por exemplo, você pode usar o programa *primeiro* para exibir o conteúdo de um número ilimitado de arquivos sem ter que modificar o código do programa. Felizmente, C facilita para seus programas o suporte aos argumentos da linha de comando. Cada vez que você chama um programa C, o sistema operacional

passa cada argumento de linha de comando para o programa como um parâmetro para a função *main*. Para acessar os argumentos da linha de comandos, você precisa declarar *main* como segue:

```
void main int(int argc, char *argv[])
  {
    // comandos do programa
  }
```

O primeiro parâmetro, *argc*, contém o número de entradas distintas na linha de comando. Considere a linha de comando a seguir:

```
C:\>  PRIMEIRO  10  NOMEARQ.EXT   <Enter>
```

Após a chamada desta linha de comando, o parâmetro *argc* conterá o valor 3. Como o valor que C atribui a *argc* inclui o nome do comando, *argc* sempre conterá um valor maior ou igual a 1. O segundo parâmetro, *argv*, é uma matriz de ponteiros para as strings de caracteres que apontam para cada argumento de linha de comando. Dada a linha de comando anterior, os elementos da matriz *argv* serão ponteiros para o seguinte:

```
argv[0]  contém um ponteiro para "PRIMEIRO.EXE"
argv[1]  contém um ponteiro para "10"
argv[2]  contém um ponteiro para "NOMEARQ.EXT"
argv[3]  contém NULL
```

Vários programas neste livro utilizam muito os argumentos na linha de comando.

EXIBINDO UM CONTADOR DE ARGUMENTOS DA LINHA DE COMANDOS 670

Cada vez que você chama um programa C, o sistema operacional passa o número de argumentos de linha de comando — bem como ponteiros para os elementos reais — para a função *main*. O programa C a seguir, *cnt_cmd.c*, usa o parâmetro *argc* para exibir o número de argumentos de linha de comando passados para o programa:

```
#include <stdio.h>

void main(int argc, char *argv[])
  {
    printf("Número de entradas na linha de comando: %d\n", argc);
  }
```

Assumindo que você chame *cmd_cnt* sem parâmetros, *cmd_cnt* exibirá o seguinte:

```
C:\>  CNT_CMD   <Enter>
O número de entradas na linha de comando é 1
C:\>
```

Se você incluir os argumentos *A*, *B* e *C*, *cnt_cmd* exibirá o seguinte:

```
C:\>  CNT_CMD  A B C <Enter>
O número de entradas na linha de comando é 4
C:\>
```

EXIBINDO A LINHA DE COMANDO 671

Como aprendeu, cada vez que você chama um programa C, o sistema operacional passa o número de argumentos de linha de comando — bem como ponteiros para os elementos reais — para a função *main* como parâmetros. O programa a seguir, *exib_cmd.c*, usa o parâmetro *conta* dentro de um laço *for* para exibir cada uma das entradas da linha de comando:

```
#include <stdio.h>
```

```
void main(int argc, char *argv[])
 {
   int i;
   for (i = 0; i < argc; ++i)
     printf ("argv[%d] aponta para %s\n", i, argv[i]);
 }
```

Se você chamar *exib_cmd* sem parâmetros, o programa exibirá o seguinte:

```
C:\> EXIB_CMD    <Enter>
argv[0]  aponta para C:\EXIB_CMD.EXE
C:\>
```

Da mesma forma, se você chamar *exib_cmd* com os argumentos *A*, *B* e *C* da linha de comandos, o programa exibirá o seguinte:

```
C:\> EXIB_CMD  A B C   <Enter>
argv[0]  aponta para C:\EXIB_CMD.EXE
argv[1]  aponta para A
argv[2]  aponta para B
argv[3]  aponta para C
C:\>
```

672 TRABALHANDO COM ARGUMENTOS DA LINHA DE COMANDO ENTRE ASPAS

Toda vez que você chama um programa C, o sistema operacional passa para *main* como parâmetros o número de entradas na linha de comando e uma matriz de ponteiro para as entradas reais. Algumas vezes seus programas precisam trabalhar com parâmetros que o sistema operacional passa a partir da linha de comando dentro de aspas. Por exemplo, assuma que um programa chamado *achatxt* pesquise um texto específico em um arquivo especificado pelo usuário, como mostrado aqui:

```
C:\> ACHATXT "Os membros da"  NOMEARQ.EXT   <Enter>
```

A maioria dos compiladores trata os parâmetros entre aspas como um único argumento. Experimente o programa *exib_cmd*, apresentado na dica anterior para determinar como seu compilador trata os parâmetros entre aspas:

```
C:\> EXIB_CMD  "Os membros da"  NOMEARQ.EXT  <Enter>
argv[0]  aponta para C:\EXIB_CMD.EXT
argv[1]  aponta para Os membros da
argv[2]  aponta para NOMEARQ.EXT
C:\>
```

673 EXIBINDO O CONTEÚDO DE UM ARQUIVO A PARTIR DA LINHA DE COMANDO

Várias dicas anteriores mostraram como usar os parâmetros *argc* e *argv* para acessar os parâmetros da linha de comando. *O programa a seguir, exib_arq.c, usa argv para exibir o conteúdo do arquivo que a linha de comando* especificou:

```
#include <stdio.h>

 void main(int argc, char *argv[])
 {
    FILE *pa;          // Ponteiro do arquivo

    char linha[255];   // Linha do arquivo

    if ((pa = fopen(argv[1], "r")) == NULL)
```

```
        printf("Erro ao abrir %s\n", argv[1]);
    else
      {
        // Lê e exibe cada linha do arquivo
        while (fgets(linha, sizeof(linha), pa))
           fputs(linha, stdout);
        fclose (pa);   // Fecha o arquivo
      }
}
```

Para exibir o conteúdo de um arquivo, chame *exib_arq* com o nome de arquivo que você quer, como mostrado aqui:

```
C:\>  EXIB_ARQ  NOMEARQ.EXT   <Enter>
```

Observe o comando *if* que abre o arquivo especificado na linha de comando. A chamada da função *fopen* dentro do comando *if* tenta abrir o arquivo que *argv[1]* especificou. Se o arquivo não existir, *fopen* retornará *NULL*, e o programa exibirá uma mensagem dizendo que não poderá abrir o arquivo. Se o usuário não especificar um nome de arquivo, então *argv[1]* conterá *NULL*, que também fará *fopen* retornar *NULL*. Se *fopen* abrir o arquivo com sucesso, então o programa usará uma laço *while* para ler e exibir o conteúdo do arquivo.

TRATANDO ARGV COMO UM PONTEIRO 674

Muitas dicas anteriores usaram a matriz de ponteiros *argv* para acessar os argumentos da linha de comando. Como *argv* é uma matriz, seus programas podem usar um ponteiro para acessar seus elementos. Se você usar um ponteiro para acessar os elementos de *argv*, *argv* se tornará um ponteiro para uma matriz de ponteiros. O programa a seguir, *argv_ptr.c*, tratará *argv* como um ponteiro para um ponteiro, depois usará *argv* para exibir a linha de comando:

```
#include <stdio.h>

void main(int argc, char **argv)
 {
   while (*argv)
      printf ("%s\n", *argv++);
 }
```

Observe a declaração de *argv* do programa como um ponteiro para um ponteiro para uma string de caracteres. O programa usa um comando *while* para percorrer em um laço os argumentos da linha de comando até que o valor para o qual **argv* aponta seja *NULL*. Como você se lembrará, C usa *NULL* para indicar o último argumento da linha de comando. Dentro do laço *while*, o comando *printf* exibirá uma string para a qual *argv* aponta. *Printf* então incrementa o valor em *argv* para que aponte para o próximo argumento da linha de comando.

COMPREENDENDO COMO C SABE SOBRE A LINHA DE COMANDO 675

Toda vez que você executa um programa, o sistema operacional carrega o programa para a memória. No caso do DOS, o sistema operacional carrega primeiro 256 bytes na memória chamado *prefixo de segmento de programa*, que contém informações, tais como a tabela de arquivos do programa, segmento do ambiente e linha de comando. A Figura 675 ilustra o formato do prefixo de segmento de programa do DOS.

Como você pode ver, começando no deslocamento 80H, o DOS armazena até 128 bytes de informações da linha de comando. Quando você compila um programa C, o compilador C insere código adicional que analisa as informações da linha de comando, atribuindo o código à matriz *argv*, que torna os argumentos fáceis para você acessar dentro de seus programas.

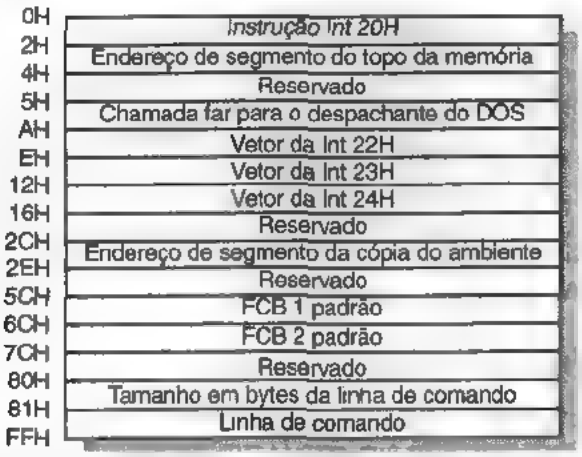

Figura 675 Prefixo de segmento de programa do Dos.

676 COMPREENDENDO O AMBIENTE

Como você sabe, tanto o DOS quanto o UNIX armazenam informações em uma região de memória chamada ambiente. Usando o comando SET, você pode exibir, acrescentar ou alterar as variáveis do ambiente. Dependendo da função do seu programa, algumas vezes você precisa acessar informações que o ambiente contém. Por exemplo, muitos programas usam a variável de ambiente TEMP para determinar a unidade de disco e o subdiretório dentro do qual o programa deve criar arquivos temporários. C facilita o acesso ao conteúdo das variáveis do ambiente. Um modo de acessar o ambiente é declarar *main*, como mostrado aqui:

```
void main(int argc, char *argv[], char *env[])
```

Exatamente como C lhe permite usar uma matriz de ponteiros de string de caracteres para acessar os argumentos de linha de comando de um programa, você pode acessar as variáveis de ambiente de um modo similar. O programa a seguir, *exib_amb.c*, usa a matriz *env* para exibir as entradas de ambiente atuais:

```
#include <stdio.h>

void main(int argc, char *argv[], char *env[])
 {
   int i;

   for (i = 0; env[i] != NULL; i++)
     printf ("env[%d] aponta para %s\n", i, env[i]);
 }
```

Como você pode ver, o programa percorre em um laço as entradas da matriz *env* até encontrar um valor *NULL*, o que indica para o programa que ele encontrou o final do ambiente.

677 TRATANDO ENV COMO UM PONTEIRO

Na dica anterior você aprendeu que C lhe permite usar a matriz *env* de ponteiros para strings de caracteres para acessar o conteúdo do ambiente. Como *env* é uma matriz, você pode tratar *env* como um ponteiro. O programa a seguir, *amb_ptr.c*, tratará *env* como um ponteiro para um ponteiro para uma string de caracteres, então usa *env* para exibir o conteúdo do ambiente:

```
#include <stdio.h>

void main(int argc, char **argv, char **env)
 {
   while (*env)
     printf("%s\n", *env++);
 }
```

Como você pode ver, o programa repete um laço até que *env* aponte para *NULL*, o que indica o fim do ambiente. Dentro do laço, o comando *printf* imprime a string para a qual *env* aponta e depois incrementa *env* para apontar para a próxima entrada.

USE VOID PARA OS PARÂMETROS DE MAIN 678

Quando seu programa não usa parâmetros de linha de comando e você não precisa usar *argc* e *argv*, você pode omitir os parâmetros e declarar *main* como:

```
void main()
```

No entanto, quando uma função não recebe parâmetros, você deve usar a palavra-chave *void* para tornar absolutamente claro para o leitor que a função não recebe parâmetros, como mostrado aqui:

```
void main(void);
```

TRABALHANDO COM NÚMEROS DA LINHA DE COMANDO 679

À medida que você criar programas que usam argumentos de linha de comando, eventualmente precisará trabalhar com números na linha de comando. Por exemplo, a linha de comando a seguir instrui o programa *primeiro* a exibir as primeiras 15 linhas do arquivo *autoexec.bat*.

```
C:\> PRIMEIRO 15 AUTOEXEC.BAT <Enter>
```

Quando a linha de comando contém números, a matriz *argv* armazena os números no formato ASCII. Para usar o número, você precisa primeiro converter o número de ASCII para um valor inteiro ou em ponto flutuante. Para converter o número, use as funções *atoi*, *atol* e *atof*, sobre as quais você aprendeu na seção Matemática. O programa a seguir, *alarmes.c*, soa o alto-falante interno do computador o número de vezes especificado na linha de comando. Por exemplo, a linha de comando a seguir instrui *alarmes* a soar três vezes o alto-falante.

```
C:\> ALARMES 3 <Enter>
```

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   int conta; // Número de vezes a soar o alto-falante
   int i;     // Número de vezes que o alto-falante soou

   // Determina o número de vezes a soar o aviso sonoro
   conta = atoi(argv[1]);
   for (i = 0; i < conta; i++)
     putchar(7);   // ASCII 7 soa o alto-falante
 }
```

Se o usuário especificar um parâmetro de linha de comando que não é um inteiro válido, então a função atoi retornará o valor 0.

COMPREENDENDO OS VALORES DE STATUS DA SAÍDA 680

Muitos comandos do DOS suportam valores de status de saída que você pode testar a partir de dentro de seus arquivos de lote para determinar o sucesso do comando. Por exemplo, o comando XCOPY do DOS suporta os valores de status de saída listados na Tabela 680.

Tabela 680 Valores de status de saída para XCOPY.

Valor de Saída	Significado
0	Operação de cópia de arquivo bem-sucedida
1	Nenhum arquivo encontrado para copiar

Tabela 680 Valores de status de saída para XCOPY. (Continuação)

Valor de Saída	Significado
2	Cópia de arquivo terminada por CTRL+C do usuário
4	Erro de inicialização
5	Erro de escrita no disco

Usando o comando IF ERRORLEVEL, seus arquivos de lote podem testar o status de saída de um programa para determinar se o comando foi bem-sucedido, e, depois, continuar processando o arquivo de lote apropriadamente. À medida que você cria programas, pode querer fornecer suporte para o valor de status da saída. O modo mais fácil de retornar um valor de status de erro é usar a função *exit*, como mostrado aqui:

```
exit (valor_status_saida);
```

Por exemplo, a chamada de função a seguir retorna o valor de status de saída 1:

```
exit(1);
```

Quando seu programa chamar a função *exit*, ele terminará imediatamente e retornará ao sistema operacional o valor de status de saída especificado. O programa a seguir, *novotipo.c*, exibe o conteúdo de um arquivo. Se o programa não puder abrir o arquivo especificado na linha de comando, retornará o valor de status de saída 1. Se *novotipo* exibir com sucesso o conteúdo do arquivo, retornará o valor de status de saída 0:

```
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   char linha[255];  // Linha lida do arquivo
   FILE *pa;

   if (pa = fopen(argv[1], "r"))
     {
       while (fgets(linha, sizeof(linha), pa))
         fputs(linha, stdout);
       fclose(pa);
       exit(0);  // Sucesso
     }
   else
     {
       printf("Não foi possível abrir %s\n", argv[1]);
       exit (1);
     }
 }
```

Nota: Muitos compiladores C fornecem uma função chamada _exit, a qual, tal como exit, finaliza imediatamente um programa e retorna um valor de status de saída. No entanto, ao contrário de exit, que primeiro fecha os arquivos que estão abertos e grava os buffers de saída, a função _exit não fecha os arquivos abertos, o que pode resultar na perda de dados

681 Usando return para o Processamento do Status da Saída

Dentro de uma função C, o comando *return* finaliza a execução de uma função e retorna o valor especificado para a função chamadora. Dentro da função *main*, o comando *return* comporta-se similarmente ao seu desempenho dentro de uma função, finalizando a execução do seu programa e retornando o valor que o programa especifica para o sistema operacional (o chamador do programa). O programa a seguir, *ret_sai.c*, exibe o conteúdo de um arquivo na tela. Se o programa não puder abrir o arquivo, *ret_sai* retornará o status de saída 1. Se *ret_sai* exibir com sucesso o conteúdo de um arquivo, ele retornará o valor de status de saída 0:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
 {
   char linha[255];  // Linha lida do arquivo
   FILE *pa;

   if (pa = fopen(argv[1], "r"))
     {
       while (fgets(linha, sizeof(linha), pa))
          fputs(linha, stdout);
       fclose(pa);
       return(0);  // Sucesso
     }
   else
     {
       printf("Não foi possível abrir %s\n", argv[1]);
       return(1);
     }
 }
```

Observe que o programa alterou a definição de *main* para indicar que a função retornará um valor de status de saída inteiro.

DETERMINANDO SE MAIN DEVE OU NÃO SER DECLARADA COMO VOID 682

Vários programas que este livro apresenta definem a função *main* como mostrado nestas duas implementações a seguir:

```
void main(void)
void main(int argc, char *argv[])
```

A palavra-chave *void*, que aparece na frente de *main*, diz ao compilador (e aos programadores que lerem seu programa) que a função *main* não usa um comando *return* para retornar um valor de status de saída para o sistema operacional. No entanto, a palavra-chave *void* não impede que seu programa use a função *exit* para retornar um valor de status de saída. No entanto, como regra, se *main* não usa o comando *return*, você precisa usar a palavra-chave *void* para preceder o nome da função Se você não usar a palavra-chave *void*, alguns compiladores poderão emitir uma mensagem de advertência similar a esta:

```
Warning Function should return a value in function main
```

PESQUISANDO O ITEM ESPECÍFICO NO AMBIENTE 683

Na Dica 676 você aprendeu como usar a matriz de ponteiros de strings de caracteres *env* para acessar a cópia do ambiente do programa, como mostrado aqui:

```
void main(int argc, char *argv[], char *env[])
```

Quando seus programas precisam pesquisar um item específico no ambiente, você pode achar conveniente usar a função *getenv*, implementada como:

```
#include <stdlib.h>

char *getenv(const char *nome_item);
```

A função *getenv* pesquisa entre os itens do ambiente um item específico, tal como "TEMP". O nome do item não precisa incluir o sinal de igual. Se o item especificado estiver no ambiente, a função *getenv* retornará

um ponteiro para o valor do item. Se o programa não encontrar o item, *getenv* retornará NULL. O programa a seguir, *exibecam.c*, pesquisa o item PATH no ambiente. Se o programa encontrar o item, exibirá o valor dele:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   char *entrada;
   entrada = getenv("PATH");
   if (*entrada)
      printf("PATH=%s\n", entrada);
   else
      printf("PATH não está definido\n");
 }
```

684 Como o DOS Trata o Ambiente

Quando você trabalha com o DOS, o ambiente fornece uma região na memória na qual você pode colocar informações da configuração, tais como o caminho dos comandos ou o prompt do sistema. O comando SET lhe permite exibir, somar ou modificar os itens do ambiente, como mostrado aqui:

```
C:\>  SET   <Enter>
COMSPEC=C:\DOS\COMMAND.COM
PROMPT=$P$G
PATH=C:\DOS;C:\WINDOWS;C:\TCLITE\BIN
C:\>
```

O DOS mantém uma cópia-mestre do ambiente que o usuário pode modificar somente com SET. Quando você chama um programa, o DOS cria uma cópia do conteúdo atual do ambiente e passa a cópia para seus programas, como mostrado na Figura 684.

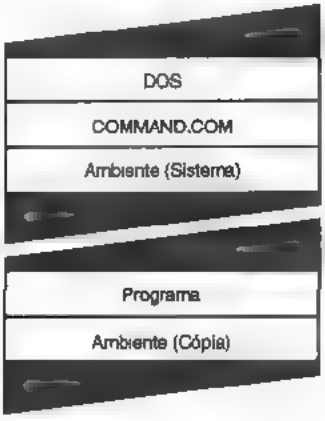

Figura 684 O DOS passa a cada programa sua própria cópia do ambiente.

Como seus programas recebem uma cópia do ambiente, as mudanças que o programa faz nos itens do ambiente não afetam o ambiente-mestre. Várias dicas nesta seção apresentam funções que lêem ou definem os itens do ambiente. Em cada caso, essas funções somente acessam a cópia do ambiente do programa.

Nota: A maioria dos programas Windows usa o registro do sistema Windows para manter as informações sobre o ambiente. Você aprenderá mais sobre o registro em dicas posteriores.

685 Usando a Variável Global environ

Na Dica 676 você aprendeu que seus programas podem usar a matriz de ponteiros de string de caracteres que o DOS passa para a função *main* do programa para acessar a cópia do ambiente fornecida pelo DOS, como mostrado aqui.

```
void main(int argc, char *argv[], char *env[])
```

Além de permitir que você use *env*, C também define uma variável global chamada *environ*, que contém a cópia do ambiente do programa. O programa a seguir, *ambiente.c*, usa a variável global *environ* para exibir o conteúdo do ambiente:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   int i;

   for (i = 0; environ[i]; i++)
     printf("%s\n", environ[i]);
 }
```

Quando seus programas usam a função *putenv* para acrescentar ou modificar um item no ambiente, você deve mais tarde acessar suas entradas no ambiente usando a função *getenv* ou acessando a variável global *environ*. Para colocar um item na cópia do ambiente do programa, *putenv* pode precisar mover a cópia do ambiente do programa, o que invalida o ponteiro *env* que o DOS passa para *main*.

ACRESCENTANDO UM ITEM AO AMBIENTE ATUAL 686

Na Dica 684 você aprendeu que o DOS mantém um cópia do ambiente-mestre. O DOS copia o ambiente-mestre e passa a cópia para cada programa que você chama. Como resultado, seus programas normalmente não podem alterar os itens no ambiente-mestre. Em vez disso, o DOS permite que quaisquer alterações que seus programas fizerem ao ambiente sejam aplicadas somente à cópia do ambiente do programa. No entanto, algumas vezes, seus programas precisam armazenar um item na cópia do ambiente. Por exemplo, assuma que um programa gere um processo-filho que precisa conhecer o nome de um arquivo específico. O programa pode primeiro colocar o nome de arquivo em sua cópia do ambiente. Quando o programa gera o processo-filho, o filho receberá uma cópia do ambiente do programa, e, portanto, poderá acessar o nome de arquivo. Para esses casos, seus programas podem usar a função *putenv*, como mostrado aqui:

```
#include <stdlib.h>

int putenv(const char *item);
```

Se *putenv* acrescentar com sucesso o item na cópia de ambiente de programa, *putenv* retornará o valor 0. Se ocorrer um erro (por exemplo, o ambiente está cheio), então *putenv* retornará -1. O programa a seguir, *putenv.c*, ilustra como usar a função *putenv*:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   if (putenv("LIVRO=Bíblia do Programador C/C++"))
      printf("Erro ao escrever no ambiente\n");
   else
     {
       int i;
       for (i = 0; environ[i]; ++i)
         printf("%s\n", environ[i]);
     }
 }
```

Nota: Não chame putenv com uma variável string de caracteres automática ou com um ponteiro string que o código do seu programa possa liberar. Além disso, se seu programa usa a matriz de ponteiros de string de caracteres que o DOS passa para main, certifique-se de que a função putenv possa mover a cópia do ambiente do programa, invalidando, desse modo, o ponteiro env. Para acessar os itens do ambiente, use getenv ou a variável global environ.

687 ACRESCENTANDO ELEMENTOS NO AMBIENTE DOS

Você aprendeu que, ao rodar um programa, o DOS copia os itens do ambiente atuais e passa a cópia para o programa. Como o programa não tem acesso à cópia-mestre dos itens do ambiente, o DOS não pode alterar as entradas que o DOS mantém. Como o DOS não fornece proteção de memória que venha a impedir um programa de acessar a memória de outro programa, você pode escrever um programa que localiza e encontra a cópia-mestre do ambiente do DOS. Após conhecer a localização do ambiente, é possível alterar ou excluir entradas existentes ou acrescentar novos itens ao ambiente. Como o ambiente tem um tamanho fixo, algumas vezes as entradas que você quer acrescentar não caberão no ambiente. Em alguns casos, você poderá alterar o tamanho do ambiente e alocar mais memória para evitar ficar sem espaço.

O processo de localizar e ajustar o tamanho do ambiente-mestre do DOS é muito complexo. No entanto, se você tiver um aplicativo que precisa alterar as entradas, pode aprender como fazer isso. No livro *DOS Programming: The Complete Reference*, Osborne/McGraw-Hill, 1991, Kris Jamsa dedica um capítulo inteiro ao trabalho com o ambiente. Como a discussão e programas de exemplo poderiam requerer de 20 a 30 páginas e estão fora do escopo deste livro, você simplesmente deve saber que é possível para seus programas atualizar a cópia-mestre do ambiente. Se você quiser atualizar o ambiente-mestre, consulte o livro do Jamsa sobre programação no DOS.

688 ABORTANDO O PROGRAMA ATUAL

À medida que seus programas vão se tornando mais complexos, algumas vezes, se um erro crítico ocorrer, você poderá querer que o programa termine imediatamente e exiba uma mensagem de erro em *stderr*. Nesses casos, seus programas poderão usar a função *abort*, como mostrado aqui:

```
#include <stdlib.h>

void abort(void);
```

Quando seu programa chamar a função *abort*, ela exibirá a seguinte mensagem em *stderr* e depois chamará a função *_exit* do DOS com o valor de status de saída 3:

```
Abnormal program termination
```

O melhor modo de compreender a função *abort* é considerar a seguinte implementação:

```
void abort(void)
 {
   fputs("Término anormal do programa", stderr);
   _exit(3);
 }
```

É importante observar que *abort* chama *_exit* e não *exit*. Como já discutido na Dica 680, a função *_exit* não fecha os arquivos abertos nem grava qualquer saída armazenada nos buffers de memória.

689 DEFININDO FUNÇÕES QUE EXECUTAM AO TÉRMINO DO PROGRAMA

Algumas vezes você pode querer que seu programa execute automaticamente uma ou mais funções quando ele terminar. Para fazer isso, seu programa pode usar a função *atexit*, que permite que seu programa especifique até 32 funções que seu programa executa automaticamente ao terminar, como mostrado aqui:

```
#include <stdlib.h>

int atexit(void (*funcao(void));
```

As funções chamadas por *atexit* não podem usar parâmetros. Se as funções precisarem acessar dados específicos, então você precisará declarar os dados como variáveis globais. Quando você definir uma lista de funções de final, seu programa executam as funções na ordem, começando com a última função que você registrou e terminando com a primeira função que você registrou. O programa a seguir, *atexit.c*, usa a função *atexit* para registrar duas funções de final:

```
#include <stdio.h>
#include <stdlib.h>

void primeira(void)
 {
   printf("Primeira função registrada\n");
 }

void segunda(void)
 {
   printf("Segunda função registrada\n");
 }

void main(void)
 {
   atexit(primeira);
   atexit(segunda);
 }
```

Quando você compilar e executar o programa atexit.c, sua tela exibirá a seguinte saída:

```
Segunda função registrada
Primeira função registrada
C:\>
```

Como você pode ver, as funções na lista de final executam na ordem oposta em que o programa registra as funções (em outras palavras, a última função registrada é a primeira função executada).

Nota: Se seu programa chama a função exit, o programa executa as funções na lista de final. No entanto, se o programa chama _exit, então o programa não executa as funções na lista de final.

COMPREENDENDO AS BIBLIOTECAS 690

Muitas dicas aqui discutem as funções da biblioteca de execução de C. Quando você usa uma função da biblioteca de execução no seu programa, o linkeditor carrega o código correspondente de um arquivo de biblioteca para seu programa executável. A vantagem óbvia de usar as funções da biblioteca de execução é que você não precisa escrever o código correspondente. Se você examinar os arquivos que acompanham seu compilador, verá que muitos arquivos têm a extensão LIB. Esses arquivos contêm bibliotecas de objeto. Quando você compila e vincula seus programas, o linkeditor examina os arquivos LIB para solucionar as referências às funções.

À medida que você cria funções úteis, pode criar suas próprias bibliotecas. É possível então, rapidamente, usar uma função que você criou para um programa dentro de outro programa. A maioria dos compiladores fornece um programa gerenciador que lhe permite criar e modificar as bibliotecas. Embora os programas para o *Turbo C++ Lite* não possam criar bibliotecas, o programa de biblioteca do *Turbo C++* da Borland chama-se *TLI*, o gerenciador do Microsoft *C/C++* chama-se *LIB*; e os compiladores C++ para o Windows de ambas as companhias suportam bibliotecas. Várias dicas a seguir discutem as operações de biblioteca.

REUTILIZANDO O CÓDIGO-OBJETO 691

À medida que você for criando funções, freqüentemente verá que uma função que escreveu para um programa servirá para outro programa. Por exemplo, o programa *str_tam.c* (localizado no CD-ROM que acompanha este livro) contém a função *string_tamanho*, como mostrado aqui:

```
int string_tamanho(char *str)
 {
   int tamanho = 0;

   while (*str++)
      tamanho++;
   return(tamanho);
 }
```

Compile o arquivo para criar o arquivo-objeto *str_tam.obj*. Após você compilar o arquivo-objeto, escreva o programa *enc_tam.c*, como mostrado aqui, que usa a função para exibir o tamanho de várias strings diferentes:

```
#include <stdio.h>

int tamanho_string(char *);

void main(void)
 {
   char *titulo= "Bíblia do Programador C/C++";
   char *secao = "Ferramentas";

   printf("O tamanho de %s é %d\n", titulo, tamanho_string(titulo));
   printf("O tamanho de %s é %d\n", secao, tamanho_string(secao));
 }
```

Se você estiver usando o *Turbo C++ Lite*, compile o programa como segue para que ele use o conteúdo do arquivo-objeto *str_tam.obj* (você também pode criar um projeto e acrescentar ambos os arquivos C ao projeto, como já detalhou a Dica 268).

```
C:\>  TC  ENC_TAM.C  STR_TAM.OBJ   <Enter>
```

Neste exemplo, você pode combinar o código-objeto da função com o seu programa para solucionar e usar a função. Como você aprenderá na dica a seguir, entretanto, compilar arquivos-objetos desse modo tem utilidade limitada.

692 PROBLEMAS COM A COMPILAÇÃO DE ARQUIVOS C E OBJ

Na dica anterior vimos que, para reusar as funções, você pode compilar a função separadamente para criar um arquivo OBJ, e, mais tarde, compilar o código C do programa e o arquivo OBJ da função para criar um programa executável, como mostrado aqui:

```
C:\>  TC  UM_ARQ.C  FUNCAO.OBJ   <Enter>
```

Embora essa técnica lhe permita solucionar e usar o código da função, ela também restringe o número de funções que você pode solucionar para o tamanho da sua linha de comando. Por exemplo, assuma que seu programa use 10 funções que residem em arquivos-objetos separados. Você verá que lembrar quais arquivos precisa compilar — e até mesmo incluir todos os nomes de arquivo na linha de comando — poderá se tornar muito difícil. Uma solução é agrupar todas as suas funções em um único arquivo-objeto. No entanto, uma melhor solução é criar um arquivo de biblioteca que contenha o código-objeto para cada função. O arquivo de biblioteca é preferível ao arquivo OBJ porque, como você aprenderá, a maioria dos arquivos gerenciadores de biblioteca lhe permite atualizar rapidamente a biblioteca (substituindo, acrescentando ou excluindo arquivos-objetos).

693 CRIANDO UM ARQUIVO DE BIBLIOTECA

Dependendo do seu compilador, o nome do seu gerenciador de biblioteca e as opções da linha de comando que o programa oferece serão diferentes. No entanto, a lista a seguir detalha as operações que a maioria dos programas gerenciadores de biblioteca suporta:

1. Criar uma biblioteca
2. Acrescentar um ou mais arquivos-objetos à biblioteca
3. Substituir um arquivo de código-objeto com outro
4. Excluir um ou mais arquivos-objetos da biblioteca
5. Listar as rotinas que a biblioteca contém

Por exemplo, o comando a seguir usa o gerenciador TLIB da Borland para criar uma biblioteca chamada *minhabib.lib* e inserir na biblioteca o código-objeto para a função *string_tamanho* que *str_tam.obj* contém:

```
C:\>  TLIB  MINHABIB.LIB +STR_TAM.OBJ   <Enter>
```

Após o arquivo de biblioteca existir, você pode usar a biblioteca para compilar e ligar o programa *enc_tam.c*, como mostrado aqui:

```
C:\> TC ENC_TAM.C MINHABIB.LIB <Enter>
```

COMPREENDENDO AS OPERAÇÕES DA BIBLIOTECA COMUM 694

Dependendo do seu compilador, as operações que sua biblioteca suporta podem ser diferentes. No entanto, a maioria dos gerenciadores lhe permite acrescentar e remover arquivos de código-objeto, usando o sinal de adição (+) para acrescentar um arquivo e o sinal de subtração (-) para remover um arquivo-objeto. A Tabela 694 lista várias operações de biblioteca que usam o arquivo *minhabib.lib*.

Tabela 694 Operações comuns com bibliotecas.

Comando	Operação
minhabib.lib +strcpy.obj	Acrescenta o arquivo-objeto *strcpy obj* na biblioteca
minhabib.lib ++strtam.obj+strmais.obj	Acrescenta os arquivos-objetos *strtam.obj* e *strmais.obj* na biblioteca
minhabib.lib -strmin.obj	Remove o arquivo-objeto *strmin.obj* da biblioteca
minhabib.lib -strmin.obj+strtam.obj	Remove o arquivo-objeto *strmin.obj* da biblioteca, e, ao mesmo tempo, acrescenta o arquivo-objeto *strtam.obj*
minhabib.lib +-strmin.obj	Substitui o arquivo-objeto *strmin.obj* na biblioteca pelo arquivo atual no disco
*minhabib.lib *strmais.obj*	Extrai o código do arquivo-objeto *strmais.obj* da biblioteca e cria um arquivo com o mesmo nome no disco

LISTANDO AS ROTINAS EM UM ARQUIVO DE BIBLIOTECA 695

Como você aprendeu, os arquivos de biblioteca fornecem locais convenientes de armazenagem para as funções que você poderá querer usar em outros programas. Dependendo do seu compilador, as operações que o gerenciador de bibliotecas oferece diferirão. No entanto, a maioria dos gerenciadores permitirá que você veja as rotinas que um arquivo de biblioteca contém. Por exemplo, usando o TLIB do Borland C++, o comando a seguir lista as rotinas contidas no arquivo de biblioteca *graficos.lib:*

```
C:\> TLIB \BORLANDC\LIB\GRAFICOS.LIB, CON <Enter>
```

Para imprimir os nomes das funções que a biblioteca contém, substitua *CON* com *PRN* na linha de comando anterior.

USE BIBLIOTECAS PARA REDUZIR SEU TEMPO DE COMPILAÇÃO 696

À medida que seu programa aumentar de tamanho, o tempo de compilação também será maior. Um modo de você reduzir o tempo de compilaçao é extrair as funções funcionais do programa e colocá-las em uma biblioteca. Desse modo, quando mais tarde você compilar seu programa, não gastará tempo recompilando as funções. Dependendo do número de funções que o programa contém, remover as funções desse modo pode reduzir significativamente o tempo de compilação. Além disso, quando você remover o código da função do seu programa, este ficará menor, mais fácil de gerenciar e, possivelmente, mais fácil de entender.

APRENDENDO MAIS SOBRE AS CAPACIDADES DA SUA BIBLIOTECA 697

As dicas que você acabou de ler ofereceram apenas uma introdução ao assunto bibliotecas. A documentação que acompanha seu compilador discutirá as capacidades da biblioteca em detalhe. Por exemplo, chamar o TLIB da Borland sem qualquer parâmetro fará o gerenciador exibir suas opções, como mostrado aqui:

```
C:\> TLIB <Enter>
```

Da mesma forma, o LIB, da Microsoft, lhe permite usar o seguinte comando para exibir as opções disponíveis na linha de comando:

```
C:\> LIB /? <Enter>
```

698 COMPREENDENDO O LINKEDITOR

Como você aprendeu, o compilador converte seu arquivo C em linguagem de máquina. Caso seu programa chame funções que estão em bibliotecas ou em outros arquivos-objetos, o linkeditor carregará o código correspondente para determinar os endereços das chamadas das funções. Após resolver todas as funções (tanto internas quanto externas) que o programa chamar, o compilador produzirá o arquivo executável. Por exemplo, assuma que seu programa contenha os seguintes comandos:

```
void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

Quando o compilador compilar o programa, ele verá que o código chama a função *printf* mas não a define. O linkeditor, por sua vez, localiza a função *printf* na biblioteca de execução, carrega o código no seu arquivo executável, e, depois, atualiza a chamada da função para referenciar o endereço correto da função dentro do seu programa. Se o linkeditor não puder localizar uma função, ele exibirá uma mensagem de erro na sua tela dizendo que encontrou *uma referência externa não resolvida*. Portanto, a função principal do linkeditor é vincular todas as partes do código do seu programa. No entanto, dependendo do seu linkeditor, você também pode usá-lo para produzir um *mapa* do *linkeditor* que descreva o layout do seu arquivo executável, para especificar um tamanho de pilha ou para controlar o uso do segmento subjacente do programa. Para conhecer os detalhes específicos das capacidades do seu linkeditor, consulte a documentação do seu compilador.

699 VENDO AS CAPACIDADES DO LINKEDITOR

Como você aprendeu na dica anterior, o papel principal do linkeditor é juntar todas as funções que um programa usa e criar um arquivo executável. Dependendo do seu compilador, seu linkeditor poderá ter capacidades adicionais. Caso esteja usando o *Turbo C++ Lite* ou o *Turbo C++*, você pode listar as opções da linha de comando do TLINK chamando-o sem nenhum parâmetro, como mostrado aqui:

```
C:\> TLINK <Enter>
```

Se você estiver usando o Microsoft C/C++, poderá exibir as opções de linha de comando para o LINK como mostrado aqui:

```
C:\> LINK /? <Enter>
```

700 USANDO UM MAPA DO LINKEDITOR

Como você aprendeu, o linkeditor localiza as funções externas dentro do seu programa, carrega as funções no arquivo executável e atualiza os endereços de cada referência de função. Quando você depurar um programa, algumas vezes saberá que o programa tem um erro em uma posição de memória específica. Usando um *mapa do linkeditor*, que mostra onde o linkeditor carregou cada função, você poderá determinar a posição do erro. Dependendo do seu linkeditor, os passos que você precisará seguir para criar um mapa poderão ser diferentes. Embora o *Turbo C++ Lite* não permita a criação de um mapa do linkeditor, você pode usar a linha de comando a seguir com o Borland *Turbo C++* para criar um arquivo de mapa chamado *enc_tam map*:

```
C:\> BCC -lm ARQ_TAM.C STR_TAM.OBJ <Enter>
```

O arquivo de mapa é um arquivo ASCII cujo conteúdo você pode exibir ou imprimir.

USANDO ARQUIVOS DE RESPOSTA DO LINKEDITOR 701

Como você aprendeu, embora o compilador *Turbo C++ Lite* não suporte os comandos específicos do linkeditor, a maioria dos outros compiladores suporta. Quando você usa os comandos TLINK ou LINK da Borland, os formatos gerais dos comandos são:

```
TLINK [opções] arqs_obj, arq_exe, arq_mapa, arq_bib, arq_def
LINK [opções] arqs_obj, arq_exe, arq_mapa, arq_bib, arq_def
```

Dependendo do número de arquivos que você esteja vinculando, suas linhas de comando podem se tornar extremamente longas. Ambos os linkeditores permitem o uso de *arquivos de resposta* para que você não precise lembrar todos os nomes de arquivo nem o formato do comando. Um arquivo de resposta é um arquivo ASCII que contém os nomes de arquivo que você quer que o linkeditor use para cada opção. Os nomes de arquivo para cada tipo de arquivo precisam aparecer na ordem da linha de comando, com cada tipo de arquivo em uma linha diferente. Por exemplo, considere o seguinte comando TLINK:

```
C:\> TLINK ENC_STR.OBJ STR_TAM.OBJ, ENC_STR.EXE, ENC_STR.MAP, UMABIB.LIB
```

Seu arquivo de resposta neste caso conteria o seguinte:

```
ENC_STR.OBJ  STR_TAM.OBJ
ENC_STR.EXE
ENC_STR.MAP
UMABIB.LIB
```

Assumindo que você chame o arquivo de resposta de *enc_str.lnk*, pode chamar o TLINK como segue:

```
C:\> TLINK @ENC_STR.LNK     <Enter>
```

Se você estiver vinculando múltiplos arquivos-objetos que não podem caber em uma linha, seu arquivo de resposta poderá continuar em uma segunda linha. Para indicar a continuação, coloque o sinal de adição (+) no final da primeira linha.

SIMPLIFICANDO A CRIAÇÃO DE APLICATIVOS COM O MAKE 702

À medida que seus programas se tornarem mais complexos, eles normalmente irão requerer arquivos de cabeçalho específicos, módulos de código-fonte, arquivos de código-objeto e bibliotecas. Quando você fizer uma modificação no seu programa, algumas vezes será difícil lembrar quais arquivos a modificação afeta. Para simplificar sua tarefa de reconstruir um arquivo executável após fazer modificações nos seus programas, muitos compiladores C oferecem um utilitário MAKE. (O utilitário MAKE está embutido dentro do *Turbo C++ Lite*.)

MAKE é uma ferramenta de programação poderosa que trabalha com um arquivo específico do aplicativo. Esse arquivo (chamado makefile) especifica os diferentes arquivos que o compilador usará para criar um aplicativo, e lista os passos que o compilador precisará seguir quando você modificar o programa. Como veremos, os arquivos que você fornece para o MAKE são quase como programas (isto é, eles contêm condições e instruções que o MAKE avalia e executa). Existem dois modos comuns de usar o MAKE. Primeiro, você pode colocar as operações que deseja que o MAKE execute em um arquivo chamado MAKEFILE. Em seguida, você simplesmente chama o MAKE, como mostrado aqui, ou carrega o arquivo no *Turbo C++ Lite*.

```
C:\>  MAKE   <Enter>
```

Quando você usar um arquivo do MAKE, o MAKE lerá o conteúdo do arquivo e processará seus programas apropriadamente. No entanto, se você estiver trabalhando em vários programas diferentes, provavelmente irá querer criar arquivos do MAKE que usam o nome do aplicativo (o segundo método). Por exemplo, você poderia ter um arquivo chamado *enc_tam.mak*. Quando você quiser chamar MAKE com um arquivo específico, precisará incluir a opção -f, como mostrado aqui:

```
C:\>  MAKE  -f  ENC_TAM.MAK   <Enter>
```

Várias dicas a seguir descrevem as operações do MAKE em detalhe.

Nota: Quando você trabalha com pacotes de desenvolvimento em C++ para o Windows, tais como o Borland C++ 5.02 ou o Microsoft Visual C++ 5.0, o arquivo de projeto que o pacote cria gerencia o arquivo do MAKE para você.

703 CRIANDO UM ARQUIVO SIMPLES DO MAKE

O MAKE é uma ferramenta que ajuda a criar arquivos executáveis ou bibliotecas após você ter feito alterações em um arquivo usado originalmente para criar o programa ou a biblioteca. Você chama o MAKE com um arquivo que contém detalhes específicos sobre um aplicativo, tais como os arquivos usados para criar o aplicativo e as dependências do arquivo. Os arquivos do MAKE seguem um formato específico. Para começar, você especificará um arquivo-alvo e os arquivos usados para criar o alvo. Por exemplo, assuma que você queira criar o programa *orcament.exe* a partir do arquivo-fonte *orcament.c*. Dentro do seu arquivo do MAKE, você especificará a dependência, como mostrado aqui.

```
ORCAMENT.EXE:    ORCAMENT.C
```

Na linha que segue imediatamente a da dependência, você especificará o comando que o MAKE precisará executar para criar o arquivo-alvo. Neste caso, as duas linhas do arquivo do MAKE ficarão assim:

```
ORCAMENT.EXE:    ORCAMENT.C
   TC  ORCAMENT.C
```

Quando você executar o MAKE com esse arquivo, o MAKE examinará a linha de dependência. Se o arquivo especificado (neste exemplo, *orcament.exe*) não existir, ou se o arquivo-alvo for mais antigo que qualquer arquivo dos quais ele é dependente (o que significa que você alterou um dos arquivos-componentes após a última compilação), o MAKE executará o comando que segue. Neste exemplo, o MAKE chamará o compilador (*bcc orcament.c*), o que recompila o programa. Para compreender melhor este processo, crie o seguinte arquivo C, *biblia.c*:

```
#include <stdio.h>

void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

Em seguida, crie o arquivo *biblia.mak*, que contém o seguinte:

```
BIBLIA.EXE:   BIBLIA.C
   TC  BIBLIA.C
```

Use a opção -f para chamar o MAKE, como mostrado aqui, ou simplesmente carregue o arquivo *biblia.mak* no *Turbo C++ Lite*:

```
C:\> MAKE  -f  BIBLIA.MAK    <Enter>
```

Como o arquivo *biblia.exe* não existe, o MAKE executará o comando para criar o arquivo. Após o MAKE terminar, chame-o novamente usando o mesmo comando. Como o arquivo *biblia.exe* já existe, e como é mais recente que *biblia.c*, o MAKE não executará o comando. Edite o arquivo *biblia.c* e altere o comando *printf* de um modo qualquer (por exemplo, modifique o texto que é escrito na tela para "Isto é um teste"). Repita o comando MAKE. Como o arquivo *biblia.c* é mais recente que *biblia.exe*, MAKE criará o novo arquivo EXE.

704 USANDO MÚLTIPLOS ARQUIVOS DE DEPENDÊNCIA COM O MAKE

Como você aprendeu, MAKE é uma ferramenta que lhe ajuda a criar aplicativos após um ou mais arquivos-componentes de um aplicativo ser modificado. Quando você usa MAKE, algumas vezes um arquivo-alvo é dependente de vários arquivos. Por exemplo, assuma que o programa *orcament.exe* seja dependente do arquivo-fonte *orcament.c*, do arquivo de cabeçalho *orcament.h* e do arquivo de biblioteca *orcament.lib*. Dentro do MAKE, você pode especificar as dependências, como mostrado aqui:

```
ORCAMENT.EXE:    ORCAMENT.C  ORCAMENT.H  ORCAMENT.LIB
   TC  ORCAMENT.C  ORCAMENT.LIB
```

COMENTANDO SEUS ARQUIVOS DO MAKE 705

Todos os arquivos do MAKE que este livro apresentou até agora são pequenos e bem simples. À medida que a complexidade de seus arquivos do MAKE aumentar, você desejará acrescentar comentários que explicam o processamento que os arquivos executam. Para colocar um comentário dentro do arquivo do MAKE, simplesmente coloque o sinal # em qualquer ponto no seu arquivo. O MAKE considerará qualquer texto que segue (na linha atual) como um comentário. O arquivo do MAKE a seguir ilustra como usar comentários:

```
# Cria o programa ORCAMENT.EXE
# Arquivo do Make criado em 10-12-97 por Kris Jamsa

ORCAMENT.EXE:    ORCAMENT.C  ORCAMENT.H  ORCAMENT.LIB
  TC ORCAMENT.C ORCAMENT.LIB  # Contém funções contábeis
```

LINHAS DE COMANDO E MAKE 706

Como você aprendeu, se um arquivo-alvo for mais antigo que um arquivo a partir do qual você cria o arquivo-alvo, o MAKE executará um comando específico. Nos exemplos vistos até aqui, o MAKE simplesmente chama o comando BCC para compilar e vincular os arquivos correspondentes. O MAKE pode emitir qualquer comando, e, além disso, ele suporta os operadores de redirecionamento de *entrada* (<), de *saída* (>) e de *anexação* (>>). O comando a seguir, por exemplo, instrui o MAKE a compilar um programa específico, redirecionando a saída do comando para a impressora.

```
ORCAMENT.EXE:    ORCAMENT.C  ORCAMENT.H  ORCAMENT.LIB
  TC ORCAMENT.C ORCAMENT.LIB  > PRN
```

Além de suportar os operadores de redirecionamento do DOS, o MAKE suporta dois operadores especiais, << e &&. Os sinais << instruem o MAKE a redirecionar a fonte de entrada padrão do comando. No entanto, em vez de direcionar a fonte de entrada para arquivo, o MAKE usa o texto que segue imediatamente, até um delimitador que você especificar, como a entrada redirecionada. Por exemplo, o arquivo MAKE a seguir direciona o MAKE a redirecionar o texto "Bíblia do Programador C/C++" para o comando EXIBEMSG:

```
ALGUMARQ.EXE:  ALGUMARQ.C
   EXIBEARQ  << ^Bíblia do Programador C/C++
^
```

Neste caso, o comando usa o circunflexo (^) como delimitador da entrada. O MAKE permite que você use qualquer caractere, exceto o # e a barra invertida (\), como delimitador. A primeira linha que começa com o delimitador especificado marca o final do texto que você deseja redirecionar para o comando EXIBEMSG. O operador && é similar, mas não causa um redirecionamento. Em vez disso, o operador && cria um arquivo temporário que contém o texto que aparece entre os delimitadores especificados. Durante a execução do comando, o MAKE substitui o próprio operador com o nome de arquivo temporário. Você usará o operador && mais comumente para criar um arquivo de resposta do linkeditor, como mostrado aqui:

```
ALGUMARQ.EXE:    ALGUMARQ.C
   TLINK  &&^
   ALGUMARQ.C
   ALGUMARQ.EXE
   ALGUMARQ.MAP
   ALGUMARQ.LIB
^
```

Como no exemplo anterior, a primeira linha que inicia com o delimitador especificado marca o final do arquivo temporário.

Nota: No Turbo C++ Lite você não pode criar arquivos do MAKE que manipulam diretamente o linkeditor.

707 COLOCANDO MÚLTIPLAS DEPENDÊNCIAS EM UM ARQUIVO DO MAKE

Se você estiver criando um grande sistema de programas, poderá realmente ter vários arquivos executáveis diferentes. Em vez de gerenciar vários arquivos do MAKE diferentes, você pode criar um arquivo que inclua as dependências relacionadas para os programas do sistema inteiro. Por exemplo, o arquivo do MAKE a seguir contém as regras de que você precisa para criar os programas *orcament.exe, folha.exe* e *impostos.exe:*

```
ORCAMENT.EXE:    ORCAMENT.C  ORCAMENT.H
   TC  ORCAMENT.C

FOLHA.EXE:   FOLHA.C    FOLHA.H
   TC FOLHA.C

IMPOSTOS.EXE:    IMPOSTOS.C  IMPOSTOS.H
   TC IMPOSTOS.C
```

Quando você executar o MAKE com o arquivo mostrado anteriormente, o MAKE começará na primeira entrada no arquivo. Neste exemplo, se o MAKE detectar que o compilador precisa reconstruir o primeiro arquivo-alvo, o MAKE executará o comando correspondente. O MAKE então continuará executando os comandos correspondentes com o segundo e o terceiro arquivos que este exemplo especifica, como necessário.

708 REGRAS IMPLÍCITAS E EXPLÍCITAS DO MAKE

Ao criar um arquivo do MAKE, as entradas que você coloca no arquivo que informam o MAKE das dependências e das operações correspondentes do arquivo são chamadas *regras.* O MAKE suporta regras *explícitas* e *implícitas.* Uma regra explícita define um ou mais nomes-alvo, zero ou mais arquivos dependentes, e zero ou mais comandos. Os nomes de arquivo dentro de regras explícitas podem ser nomes de caminhos completos ou caracteres-chave. Todos os arquivos de exemplo do MAKE que este livro apresentou usam regras explícitas. Por outro lado, as regras implícitas são mais gerais. Uma regra implícita corresponde a todos os arquivos com uma extensão específica. O MAKE usará uma regra implícita quando você não fornecer uma regra explícita para um arquivo-alvo. Por exemplo, você poderia especificar que um arquivo OBJ depende de um arquivo C. A seguinte regra implícita instrui o MAKE a compilar todos os arquivos C cujos arquivos de código-fonte são mais recentes que o arquivo OBJ correspondente:

```
.C.OBJ:
   TC $<
```

A sintaxe para uma regra implícita é o tipo de arquivo dependente (C) seguido pelo tipo de arquivo-alvo (OBJ). A regra usa uma macro especial ($<) que, como você aprenderá, instrui o MAKE a usar o nome completo do arquivo C correspondente. Colocar apenas uma regra implícita dentro de um arquivo MAKE não tem efeito direto quando o MAKE é executado — em outras palavras, um arquivo do MAKE precisa incluir pelo menos uma regra explícita. O MAKE somente usará a regra implícita quando encontrar um arquivo-alvo para o qual você não forneceu uma regra explícita.

709 USANDO MACROS DO MAKE

Uma macro do *MAKE* é um símbolo que o MAKE substitui com um valor específico. Você pode usar macros dentro do MAKE para muitos propósitos. Por exemplo, a macro a seguir, *MODELO_MEM,* define as opções que o compilador requer para selecionar o modelo de memória small:

```
MODELO_MEM = -ms
```

Para usar o valor de uma macro dentro do seu arquivo do MAKE, coloque o nome da macro dentro de parênteses precedidos pelo sinal de cifrão. Por exemplo, a linha de comando a seguir usa a macro *MODELO_MEM:*

```
NOMEARQ.EXE:    NOMEARQ.C
   BCC $(MODELO_MEM)   NOMEARQ.C
```

Nota: Devido ao modo como o Turbo C++ Lite implementa os controles do modelo de memória, essa macro em particular não funcionaria nesse compilador — motivo pelo qual é mostrada com um comando que chama o Turbo C++.

MACROS PREDEFINIDAS DO MAKE 710

Como você aprendeu, uma macro do MAKE é um símbolo que o MAKE substitui por um valor específico. O MAKE fornece várias macros predefinidas que você pode usar dentro de seus arquivos do MAKE. Dependendo se você estiver usando uma macro dentro de uma regra explícita ou implícita, o valor que o MAKE substitui para o símbolo diferirá. A Tabela 710.1 discute como usar as macros predefinidas do MAKE nas regras explícitas. Da mesma forma, a Tabela 710.2 ilustra como usar macros em regras implícitas.

Tabela 710.1 Valores de macro predefinidos do MAKE para as regras explícitas.

Nome da Macro	Valor Retornado
$*	Nome da base dependente com o caminho
$&	Nome da base dependente sem o caminho
$.	Nome completo dependente sem o caminho
$**	Nome completo dependente com o caminho
$<	Nome completo dependente com o caminho
$?	Nome completo dependente com o caminho

Tabela 710.2 Valores de macro predefinidos do MAKE para as regras implícitas

Nome da Macro	Valor Retornado
$*	Nome da base-alvo com o caminho
$&	Nome da base-alvo sem o caminho
$.	Nome completo do alvo sem o caminho
$**	Todos os nomes de arquivo dependentes
$<	Nome completo do alvo com o caminho
$?	Todos os dependentes desatualizados

O arquivo do MAKE a seguir, por exemplo, cria uma regra implícita que informa o MAKE do relacionamento entre os arquivos com as extensões OBJ e C:

```
.C.OBJ:
   TC $<
```

Neste exemplo, o MAKE usará a extensão de regra implícita para expandir a macro $< no nome de arquivo-alvo e caminho.

EXECUTANDO PROCESSAMENTO CONDICIONAL COM O MAKE 711

Na seção Macros, anteriormente, você aprendeu como usar as diretivas do pré-processador, tais como *#if, #elif, #else* e *#endif* De um modo similar, MAKE fornece comandos de processamento condicional que iniciam com um ponto de exclamação (!), tais como *!if, !else, !elif* e *!endif*. Você também pode usar as diretivas *!ifdef, !ifndef* e *!undef* para testar as macros que você definiu e para anular a definição de uma macro. Se uma diretiva condicional for avaliada como verdadeira, MAKE executará as regras que seguem. Se a diretiva for falsa, MAKE não processará as regras correspondentes. Os comandos a seguir ilustram vários comandos condicionais diferentes:

```
!ifdef nome_macro    # Testa se a macro está definida
   # comandos
!endif

!if $(Valor) > 5 # Testa se o valor da macro Valor é > 5
   # comandos
!endif

!if ! $d(Nome_macro)  # Testa se a macro não está definida
```

```
    # comandos
!endif
```

712 TESTANDO UM NOME DE MACRO

Como você aprendeu, o MAKE lhe permite definir suas próprias macros. Dependendo do processamento que seu arquivo MAKE executa, algumas vezes você quer testar se uma determinada macro está definida. Para fazer isso, você pode usar o teste *$d(macro)*. Se a macro estiver definida, o teste retornará o valor 1. Se a macro não estiver definida, o resultado será 0.

Os comandos a seguir usam o operador condicional *!if* de MAKE para determinar se a macro *MODELO_MEM* está definida. Se a macro não estiver definida, os comandos atribuirão a ela o valor do modelo de memória small, como mostrado aqui:

```
!if ! $d(MODELO_MEM)
MODELO_MEM = -ms
!endif
```

Além de usar o teste *$d(macro)*, suas macros podem executar processamento equivalente usando os comandos condicionais *!ifdef* e *!ifndef*. Se você, mais tarde, quiser anular a definição da macro, poderá usar o comando *!undef* para fazer isso, como mostrado aqui:

```
!undef nome_macro
```

Nota: Como você aprendeu, essa atribuição em particular não funciona no arquivo do MAKE no Turbo C++ Lite, devido ao modo como esse compilador trata os modelos de memória.

713 INCLUINDO UM SEGUNDO ARQUIVO DO MAKE

Caso seus arquivos do MAKE tenham a mesma forma, então você poderia achar conveniente colocar suas regras implícitas comumente usadas em um arquivo do MAKE chamado *implicit.mak*. No início de cada um dos seus arquivos do MAKE, você pode incluir o arquivo usando a diretivas *!include* do MAKE, como mostrado aqui:

```
!include "IMPLICIT.MAK"
```

714 USANDO OS MODIFICADORES DE MACRO DO MAKE

Como você aprendeu na Dica 710, o MAKE predefine várias macros diferentes que seus arquivos podem usar para obter o arquivo-alvo ou dependente. Para dar aos seus arquivos do MAKE maior controle sobre os nomes de arquivo que essas macros retornam, o MAKE permite que você use os modificadores *B*, *D*, *F* e *R*, detalhados na Tabela 714.

Tabela 714 Os modificadores para a macro *MAKE*.

Modificador	Propósito
$(macroB)	Retorna o nome base somente
$(macroD)	Retorna a unidade e o diretório
$(macroF)	Retorna o nome base e a extensão
$(macroR)	Retorna a unidade, diretório e o nome-base

O comando a seguir, por exemplo, usa o modificador *D* com o *$<* para copiar arquivos de um diretório arquivo-alvo para um diretório BACKUP:

```
C:\SUBDIR\DICAS.EXE:    DICAS.C
    COPY  $(<D)*.C     C:\BACKUP
    TC  DICAS.C
```

Nota: Este arquivo do MAKE executa dois comandos dentro da regra. O primeiro comando copia arquivos com a extensão C para um diretório chamado BACKUP, e o segundo comando compila o arquivo-fonte.

FINALIZANDO UM ARQUIVO DO MAKE COM UM ERRO 715

Dependendo do processamento que seu arquivo do MAKE executa, algumas vezes você pode querer que o MAKE finalize seu processamento e exiba uma mensagem de erro para o usuário. Nesses casos, você pode usar a diretiva *!error*. Os comandos a seguir, por exemplo, testam se a macro *MODELO_MEM* não está definida. Se a macro não estiver definida, o MAKE exibirá uma mensagem de erro para o usuário e terminará o processamento:

```
!ifndef MODELO_MEM
!error Finalizando a criação - defina a macro MODELO_MEM
!endif
```

DESABILITANDO A EXIBIÇÃO DO NOME DO COMANDO 716

Por padrão, o MAKE exibe cada comando antes de executá-lo. Para desabilitar a exibição do comando, simplesmente preceda o nome do comando com o símbolo da arroba (@). Por exemplo, os comandos a seguir usam o símbolo @ para desabilitar a exibição do comando BCC:

```
DICAS.EXE:   DICAS.C
   @BCC  DICAS.C
```

Se você também quiser desabilitar a exibição da saída do comando, simplesmente redirecione a saída para o dispositivo NUL, como mostrado aqui:

```
DICAS.EXE:   DICAS.C
   @BCC  DICAS.C  > NUL
```

USANDO O ARQUIVO BUILTINS.MAK 717

Você já sabe que é possível usar regularmente muitas regras implícitas. Um modo de garantir que todos os seus arquivos do MAKE possam usar as regras implícitas é colocar as regras comuns em um arquivo especial chamado *builtins.mak*. Cada vez que você chama o MAKE, ele procura o arquivo *builtins.mak*. Se o arquivo existir, o MAKE processará imediatamente as informações que ele contém. Se o arquivo não existir, o MAKE continuará seu processamento usando MAKEFILE ou o arquivo que você especificou na linha de comando. O arquivo *builtins.mak* a seguir contém a regra implícita para converter arquivos C para OBJ:

```
.C.OBJ
   TC $<
```

EXECUTANDO PROCESSAMENTO DE STATUS DE SAÍDA NO MAKE 718

Quando o MAKE executa um comando, algumas vezes você pode querer que ele avalie o valor de status de saída do comando e, dependendo do resultado, continue ou termine. Se você preceder um nome de comando com um hífen seguido por um valor, o MAKE comparará o valor de status de saída de um comando com o valor que o arquivo do MAKE especifica. Se o valor de status de saída for *maior* que o valor, o MAKE abortará a criação do programa atual. Por exemplo, o comando a seguir compara o status de saída do comando *exibearq* com 3. Se o valor de saída for maior que 3, o MAKE abortará a criação, como mostrado aqui:

```
TEST.EXE:  DICAS.C
   -3 EXIBEARQ $?
```

Se você quiser que o MAKE ignore o status de saída de um comando, preceda o nome do comando com um hífen e nenhum valor correspondente:

```
   -  CC  TESTE.C
```

719 CHAMANDO E ALTERANDO UMA MACRO AO MESMO TEMPO

Como foi visto, o MAKE faz você definir suas próprias macros. Dependendo do processamento que seu arquivo do MAKE executa, algumas vezes você irá querer alterar e imediatamente usar uma macro. Por exemplo, assuma que você defina a macro *ENTRADA_ARQ* como mostrado aqui:

```
ENTRADA_ARQ = ORCAMENT.C
```

Você pode então usar a macro, como mostrado aqui:

```
ORCAMENT.EXE:    $(ENTRADA_ARQ)
   TC  $(ENTRADA_ARQ)
```

Em seguida, assuma que você queira copiar o arquivo de entrada para um arquivo com o mesmo nome, mas com a extensão SAV. Você pode alterar a macro substituindo .C por .SAV, e usando imediatamente a nova definição. O comando a seguir ilustra como fazer a cópia:

```
COPY $(ENTRADA_ARQ)    $(ENTRADA_ARQ: .C=.SAV)
```

A primeira parte do comando COPY usa o nome de arquivo *orcament.c*. A segunda parte do comando substitui .C com .SAV para criar o nome de arquivo *orcament.sav*.

720 EXECUTANDO UM COMANDO MAKE PARA MÚLTIPLOS ARQUIVOS DEPENDENTES

Você já sabe que, algumas vezes, um arquivo-alvo é dependente de dois ou mais arquivos. Dependendo do processamento que seu arquivo MAKE executa, você pode querer que ele execute um comando específico para cada arquivo. Para fazer isso, simplesmente preceda o nome do comando com um sinal &. A regra a seguir, por exemplo, instrui MAKE a compilar individualmente cada arquivo dependente desatualizado:

```
ORCAMENT.EXE:    ORCAMENT.C  CONTAS.C  FOLHA.C
   & TC $?
```

721 DETERMINANDO SE O CO-PROCESSADOR MATEMÁTICO ESTÁ PRESENTE

Caso seus programas efetuem operações matemáticas complexas, algumas vezes você poderá usar o co-processador matemático do computador para aumentar o desempenho de um programa. Para ajudar seus programas a fazer uso do co-processador matemático, existem várias bibliotecas de terceiros que fornecem funções comumente usadas. No entanto, antes que seus programas usem essas funções, eles devem verificar se o computador está equipado com um co-processador matemático. Para esses casos, muitos compiladores C definem a variável global *_8087*, que contém o valor 1 se um co-processador está presente, e 0 se não está presente. O programa a seguir, *chk_math.c*, ilustra como usar a variável global *_8087*:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   if (_8087)
     printf("Co-processador encontrado\n");
   else
     printf("Co-processador não-encontrado\n");
 }
```

Você pode usar a entrada do ambiente *87* para controlar o valor que o compilador C atribui à variável *_8087*. Para definir o valor da variável como 1, atribua a *87* o valor *Yes*, como mostrado aqui:

```
C:\> SET 87=Yes   <Enter>
```

Da mesma forma, para deixar a variável em 0, atribua ao item do ambiente o valor *No*.

Compreendendo as Macros ctype.h e istype 722

A seção Macros, anteriormente, apresentou várias macros que testam se um caractere é maiúsculo, minúsculo, alfanumérico, e assim por diante. Se você examinar o arquivo de cabeçalho *ctype.h*, encontrará definições de macro similares às seguintes:

```
#define isalpha(c) (_ctype[(c) + 1] & (_IS_UPP | _IS_LOW))
#define isascii(c) ((unsigned) (c) < 128)
#define iscntrl(c) (_ctype[(c) + 1] & _IS_CTL)
#define isdigit(c) (_ctype[(c) + 1] & _IS_DIG)
#define isgraph(c) ((c) >= 0x21 && (c) <= 0x7e)
#define islower(c) (_ctype[(c) + 1] & _IS_LOW)
```

Para reduzir o tempo de processamento que os testes de macros requerem, muitos compiladores C definem uma variável global chamada *ctype*, que contém definições para cada caractere ASCII. Usando essas definições, as macros *istype* podem usar operações bit a bit mais rápidas para efetuar o teste necessário. O programa a seguir, *ctype.c*, exibe as definições que o compilador usa para cada caractere ASCII:

```
#include <stdio.h>
#include <ctype.h>

void main(void)
 {
  int carac_ascii;

  for (carac_ascii = 0; carac_ascii < 128; carac_ascii++)
    if (isprint(carac_ascii))
      printf("Valor ASCII %d valor (hexa) %x ASCII %c\n",
        carac_ascii, _ctype[carac_ascii], carac_ascii);
    else
      printf("Valor ASCII %d valor (hexa) %x ASCII %c\n",
        carac_ascii, _ctype[carac_ascii], carac_ascii);
 }
```

Controlando o Vídeo Direto 723

O arquivo de cabeçalho *conio.h* define protótipos para as funções que efetuam E/S no console, tais como *cputs*. Para melhorar o desempenho dessas funções de E/S no console, a maioria dos compiladores C ignora o DOS e a BIOS e escreve a saída diretamente na memória de vídeo do PC. Embora a maioria das operações de vídeo seja padrão de um PC para outro, você poderá encontrar uma placa de vídeo que não suporta as operações de vídeo direto. Se você experimentar esses erros, poderá usar a variável global *directvideo* para controlar se o PC usa as rotinas de vídeo da BIOS para executar a saída do seu programa, ou se seu programa efetua E/S direta. Se você definir o valor da variável como 1, as rotinas efetuarão saída direta no vídeo. Se o valor for 0, as rotinas efetuarão sua saída usando a BIOS.

Detectando Erros Matemáticos e do Sistema 724

Várias funções da biblioteca de execução de C que este livro apresenta atribuem valores à variável global *errno* quando um erro ocorre. Quando seus programas usam essas funções, você deve testar o valor de retorno da função e o valor *errno*. A Tabela 724 define as constantes que as funções atribuem a *errno*.

Tabela 724 Valores constantes que as funções atribuem a *errno*.

Constante	Significado
E2BIG	A lista de argumento é longa demais
EACCES	Permissão negada
EBADF	Indicativo de arquivo inválido
ECONTRL	Erro nos blocos de controle de memória
ECURDIR	Tentativa de remover o diretório atual
EDOM	Um argumento viola o domínio de valores suportados
EEXIST	O arquivo já existe
EFAULT	Erro desconhecido
EINVACC	Especificador de acesso inválido
EINVAL	Valor de argumento inválido
EINVDAT	Dado de argumento inválido
EINVDRV	Especificador de unidade inválido
EINVENV	Ambiente inválido
EINVFMT	Formato de argumento inválido
EINVFNC	Número de função inválido
EINVMEM	Bloco de memória inválido especificado
ENFILE	Arquivos demais abertos
ENMFILE	Sem mais arquivos
ENODEV	Esse dispositivo não está disponível
ENOENT	Entrada inválida (arquivo ou diretório)
ENOEXEC	Erro de formato em EXEC
ENOFILE	Esse arquivo ou diretório não existe
ENOMEM	Memória insuficiente
ENOPATH	Caminho não encontrado
ENOTSAM	Não é o mesmo dispositivo
ERANGE	O resultado da função está fora do intervalo de valores válidos
EXDEV	Dispositivo de vínculo cruzado
EZERO	Erro zero

725 EXIBINDO MENSAGENS DE ERRO PREDEFINIDAS

Na dica anterior você aprendeu que várias funções matemáticas e do sistema atribuem à variável global *errno* valores de status específicos que seus programas podem ler para obter informações sobre a causa de um erro. Dependendo do processamento de seu programa, você pode querer exibir uma mensagem predefinida quando um erro ocorrer. Para ajudar seu programa a processar os erros, o compilador C fornece uma variável global chamada *sys_errlist*, que contém mensagens de erro para a maioria dos erros. Adicionalmente, para aumentar a portabilidade dos seus programas, a matriz contém mensagens de erro do ambiente Unix.

O compilador também atribui à variável global *sys_nerr* o número de mensagens de erro na matriz. O programa a seguir, *msg_erro.c*, usa a matriz *sys_errlist* para exibir as mensagens de erro predefinidas:

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
 {
   int erro;

   for (erro = 0; erro < sys_nerr; erro++)
      printf("Erro %d %s\n", erro, sys_errlist[erro]);
 }
```

DETERMINANDO O NÚMERO DE VERSÃO DO SISTEMA OPERACIONAL 726

Se você estiver desenvolvendo aplicativos para o ambiente DOS, algumas vezes seus programas precisarão saber o número da versão atual do sistema operacional. Nesses casos, seus programas podem usar as variáveis globais predefinidas *_osmajor* e *_osminor*, que contêm os números inteiro e decimal da versão. Além disso, alguns compiladores fornecem a variável *_version*. O byte inferior da constante contém o número inteiro da versão, e o byte superior contém o número decimal da versão. Por exemplo, dado o DOS 6.0, a variável *_osmajor* conterá o valor 6, e a variável *_osminor* conterá o valor 0. O programa a seguir, *versao.c*, usa as variáveis globais de versão para exibir o número da versão do sistema operacional:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   printf("Número da versão do sistema operacional %d.%d\n", _osmajor, _
osminor);
   printf("Número da versão do sistema operacional %d.%d\n",
    _version & 255, _version >> 8);
 }
```

Nota: Se o programa anterior não compilar no seu sistema, transforme em comentário os comandos que usam _version, e inclua o arquivo de cabeçalho stdlib.h.

COMPREENDENDO A PORTABILIDADE 727

Portabilidade é uma medida da facilidade com a qual você pode mover seus programas de um sistema para outro. Por exemplo, quando você escreve um programa usando a linguagem assembly para o PC, é muito difícil migrar esse programa para uma estação de trabalho que usa um linguagem assembly diferente. No entanto, se você tivesse escrito o mesmo programa em C, precisaria somente fazer algumas pequenas modificações no programa para que ele compilasse no novo sistema. À medida que você programa, lembre-se sempre da portabilidade. Normalmente você pode usar o mesmo código que escreveu para um programa para muitos outros programas, e pode poupar considerável tempo de programação e de teste se enfatizar a escrita de código portátil. Para aumentar a portabilidade do seu programa, considere o seguinte ao codificar:

- Evite os serviços do sistema operacional sempre que possível. Em lugar deles, utilize as rotinas da biblioteca de execução de C.
- Evite as funções e variáveis globais que são específicas para o seu compilador. Na maioria dos casos, o compilador precederá esses nomes de função e de variável com um sublinhado, tal como *_8087*.
- Não faça suposições sobre o tamanho dos inteiros na máquina. Por exemplo, no PC, uma variável do tipo *int* tipicamente contém 16 bits, mas, em outras máquinas, ela poderia conter 32 bits.
- Não acesse posições específicas do hardware ou dependa de interrupções específicas, a não ser que o desempenho do seu programa absolutamente requeira isso.
- Sempre tente corrigir e eliminar as mensagens de advertência do compilador.
- Não faça suposições sobre o modelo de memória que podem não existir em um ambiente baseado no Unix.
- Restrinja o código dependente do hardware ou do sistema operacional para o menor número possível de funções.

EFETUANDO UM GOTO NÃO-LOCAL 728

Na seção Introdução à Linguagem C, anteriormente você aprendeu que o comando *goto* permite que a execução do seu programa salte de um local para outro. Como você aprendeu, o rótulo para onde você quer "ir" precisa residir na função atual. Dependendo dos seus programas, algumas vezes você precisa saltar para um rótulo fora

da função atual (chamado de *goto não-local*). Para efetuar um goto não-local, seus programas podem usar as funções *setjmp* e *longjmp:*

```
#include <setjmp.h>

void longjmp(jmp_buf local, int valor_retorno);
void setjmp(jmp_buf local);
```

Para começar, seu programa usará *setjmp* para armazenar o local correto (também conhecido como *estado da tarefa*) no buffer *local*. Mais tarde, seus programas podem saltar para esse local usando *longjmp*. A primeira vez que seu programa chamar *setjmp*, a função retornará 0. Quando o programa mais tarde chamar *longjmp*, ele retornará ao local *setjmp* anteriormente armazenado, e produzirá o valor de retorno especificado dentro do parâmetro *valor_retorno* da função *longjmp*. O programa a seguir, *longjmp.c*, ilustra um goto não-local:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf local;   // Variável global

void funcao(void)
 {
   printf("Prestes a executar longjmp\n");
   longjmp(local, 1);  // Retorna 1
 }

void main(void)
 {
   if (setjmp(local) != 0)  // Salva a posição atual
     {
       printf("Retornando de longjmp\n");
       exit(1);
     }

   funcao();
 }
```

729 OBTENDO A IDENTIFICAÇÃO DO PROCESSO

Em um ambiente multitarefa, o sistema operacional atribui a cada programa um identificador exclusivo, chamado identificador do processo, ou PID. O sistema operacional Unix fornece uma função chamada *getpid*, que retorna um identificador do processo do programa. Muitos compiladores baseados no DOS fornecem uma função *getpid* similar, como mostrado aqui:

```
#include <process.h>

unsigned getpid(void);
```

Dentro do DOS, o identificador do processo é realmente o endereço do segmento do prefixo do segmento do programa (PSP). Se você tiver dois ou mais programas residentes na memória ativos, cada programa terá um identificador de processo diferente, porque cada um tem um endereço de segmento PSP exclusivo. O programa a seguir, *getpid.c*, exibe o conteúdo do identificador de processo do programa:

```
#include <stdio.h>
#include <process.h>

void main(void)
 {
   printf("Ident. do processo: %X\n", getpid());
 }
```

Nota: O Windows trata os processos e os encadeamentos de forma ligeiramente diferente que o PID do DOS. As Dicas 1376 até 1400 discutirão o gerenciamento de processos e de encadeamentos em detalhe.

CHAMANDO UM COMANDO INTERNO DO DOS 730

Várias dicas nesta seção mostraram modos como seus programas podem chamar os arquivos executáveis (EXE e COM). Dependendo do seu programa, algumas vezes você precisa chamar um comando interno do DOS ou um arquivo de lote. Nesses casos, seus programas podem usar a função *system*, como mostrado aqui:

```
#include <stdlib.h>

int system(const char *comando);
```

O parâmetro *comando* é uma string de caracteres que contém o nome do comando interno ou externo do DOS ou arquivo de lote desejado. Se *system* executar com sucesso o comando, retornará o valor 0. Se ocorrer um erro, *system* retornará o valor -1, e atribuirá à variável global *errno* um dos valores listados na Tabela 730.

Tabela 730 Os valores de ***errno*** que a função ***system*** retorna.

Valor	Significado
ENOENT	Esse arquivo não existe
ENOMEM	Não há memória suficiente
E2BIG	A lista de argumentos é grande demais
ENOEXEC	Erro no formato de *exec*

A função *system* gera uma cópia de *command.com* para executar o comando especificado. A função usa a variável de ambiente COMSPEC para localizar o processador de comandos. O programa a seguir, *system.c*, usa a função *system* e o comando *dir* do DOS para exibir uma listagem do diretório:

```
#include <stdlib.h>
#include <stdio.h>

void main(void)
 {
   if (system("DIR"))
      printf("Erro ao chamar DIR\n");
 }
```

USANDO A VARIÁVEL GLOBAL _PSP 731

Cada vez que você executa um programa, o DOS carrega o programa na memória imediatamente após um buffer de 256 bytes chamado *prefixo do segmento do programa* (PSP). O prefixo do segmento do programa contém informações sobre a linha de comando, um ponteiro para a cópia do ambiente do programa, informações da tabela de arquivos etc. A Figura 731 ilustra o conteúdo do prefixo do segmento do programa.

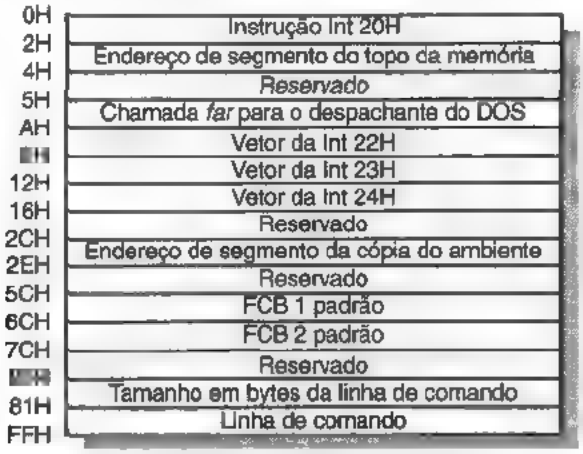

Figura 731 Conteúdo do prefixo do segmento do programa.

À medida que você começa a trabalhar em seus programas com os comandos internos do DOS, algumas vezes seus programas acessam informações que o PSP de um programa contém. O DOS, portanto, fornece um serviço do sistema que retorna o endereço do PSP. Para simplificar esses programas, alguns compiladores definem uma variável global, *_psp*, que contém o endereço do segmento do PSP do programa. O programa a seguir, *end_psp.c*, usa a variável global *_psp* para exibir o endereço de PSP do programa:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
  printf("O PSP inicia em %X\n", _psp);
 }
```

Nota: Alguns compiladores C também fornecem a função **getpsp**, *que retorna o endereço de segmento do PSP do programa, como mostrado aqui:*

```
#include <dos.h>

unsigned getpsp(void);
```

732 USANDO O MODIFICADOR CONST NAS DECLARAÇÕES DE VARIÁVEIS

Vários protótipos de funções que este livro apresenta usam a palavra-chave *const* antes dos nomes dos parâmetros, como mostrado aqui:

```
char *strcpy(char *alvo, const char *origem);
```

Ao usar a palavra-chave *const* antes de um nome de parâmetro, você diz ao compilador que o programa não deve alterar o parâmetro dentro da função. Se um comando tentar alterar o parâmetro, o compilador gerará uma mensagem de erro. C também lhe permite usar a palavra-chave *const* quando você declarar variáveis. Ao declarar uma variável como uma constante, o compilador gerará um erro cada vez que você tentar alterar a constante. Os comandos a seguir criam várias constantes diferentes:

```
const int numero = 1001;
const float preco = 39.95;
```

A vantagem de usar uma constante em vez de uma macro que você criou com *#define* é que você pode, usando a constante, especificar explicitamente o tipo do valor.

Nota: Quando você declarar uma constante, ainda poderá usar um apelido de ponteiro para alterar o valor da variável constante.

733 USANDO TIPOS ENUMERADOS

Como você aprendeu, usar nomes de variáveis significativos pode melhorar muito a legibilidade do seu programa. Além disso, substituindo valores constantes (tais como 1, 2 e 3) com nomes significativos que correspondem aos valores que as variáveis representam (tais como Segunda, Terca e Quarta), é possível aumentar a legibilidade do seu programa. Para ajudar seus programas a trabalhar com essas constantes, C suporta *tipos enumerados*. Em geral, um tipo enumerado é uma lista de itens, cada um dos quais tem um valor exclusivo. Você pode usar tipos enumerados para aumentar a legibilidade do seu programa. Por exemplo, a declaração a seguir cria um tipo enumerado chamado *dias_semana:*

```
enum dias_semana {Segunda, Terca, Quarta, Quinta, Sexta};
```

O tipo enumerado é similar à definição de uma estrutura, em que você pode declarar variáveis do tipo imediatamente, ou você poderá referenciar o nome do tipo mais tarde, como mostrado aqui:

```
enum dias_semana {Segunda, Terca, Quarta, Quinta, Sexta} dia_util;
enum dias_semana folga;
```

Após você declarar uma variável enumerada, poderá referenciar um nome de membro para atribuir um valor a uma variável, como mostrado aqui:

```
folga = Sexta;
dia_util = Terca
```

COLOCANDO UM TIPO ENUMERADO EM USO 734

Na dica anterior você aprendeu que seus programas podem usar tipos enumerados para aumentar sua legibilidade. O programa a seguir, *quedia.c*, ilustra como seus programas poderiam usar um tipo enumerado para aumentar sua legibilidade:

```
#include <stdio.h>

void main(void)
 {
   enum { Segunda, Terca, Quarta, Quinta, Sexta } dia;

   for (dia = Segunda; dia <= Sexta; dia++)
    if (dia == Segunda)
      printf("Nada de diversão - reuniões o dia todo\n");
    else if (dia == Terca)
      printf("Completar o serviço de ontem\n");
    else if (dia == Quarta)
      printf("Outro dia de trabalho duro...\n");
    else if (dia == Quinta)
      printf("Agendar as reuniões da próxima Segunda\n");
    else
      printf("Comemorar após o fim do expediente!\n");
 }
```

COMPREENDENDO UM VALOR ENUMERADO 735

Como você viu na Dica 733, cada membro dentro de um tipo enumerado tem um valor exclusivo. Por padrão, o compilador C atribui ao primeiro membro o valor 0, ao segundo, o valor 2 e assim por diante. O programa a seguir, *exibenum.c*, mostra os valores que correspondem aos dias enumerados da semana:

```
#include <stdio.h>

void main(void)
 {
   enum dias_semana {Segunda, Terca, Quarta, Quinta, Sexta};
   printf("%d %d %d %d %d\n", Segunda, Terca, Quarta, Quinta, Sexta);
 }
```

Quando você compilar e executar o programa *exibenum.c*, sua tela exibirá os valores de 0 a 4.

ATRIBUINDO UM VALOR ESPECÍFICO A UM TIPO ENUMERADO 736

Na dica anterior você aprendeu que o compilador C atribui valores exclusivos a cada membro de um tipo enumerado. Dependendo da função que seu programa executa, algumas vezes você pode querer especificar o valor de cada membro. Por exemplo, a declaração a seguir atribui os valores 10, 20, 30, 40 e 50 aos dias da semana:

```
enum dias_semana { Segunda = 10, Terca = 20, Quarta = 30,
                   Quinta = 40, Sexta = 50 };
```

O programa a seguir, *defnum.c*, atribui esses valores aos membros do tipo enumerado, e, depois, exibe os valores de cada membro:

```
#include <stdio.h>

void main(void)
```

```
{
  enum dias_semana { Segunda = 10, Terca = 20, Quarta = 30,
                     Quinta = 40, Sexta = 50 };

  printf("%d %d %d %d %d\n", Segunda, Terca, Quarta, Quinta, Sexta);
}
```

Além de atribuir um valor a cada membro, você também poderá atribuir um valor a um membro específico. O compilador C incrementará o valor de cada membro restante em 1. Por exemplo, a declaração a seguir atribui os valores 10, 11, 12, 13 e 14 aos dias da semana:

```
enum dias_semana {Segunda = 10, Terca, Quarta, Quinta, Sexta};
```

737 SALVANDO E RESTAURANDO OS REGISTRADORES

Muitos compiladores lhe permitem acessar os valores dos registradores a partir de dentro de seus programas. Os programas que efetuam essas operações normalmente colocam os valores dos registradores na pilha antes de o programa alterar o registrador, e, depois, remove o valor para restaurá-lo no registrador. Se você tem uma função que executa essas operações de baixo nível, pode instruir o compilador C a inserir instruções PUSH e POP no código-objeto. PUSH e POP automaticamente salvarão todos os registradores quando o programa chamar a função, e, mais tarde, restaurarão os registradores antes de a função terminar. Para instruir o compilador a efetuar essas operações, simplesmente inclua o modificador *_saveregs* no cabeçalho da função, como mostrado aqui:

```
int _saveregs alguma_funcao(int parametro);
```

Para compreender melhor o processamento que o modificador *_saveregs* instrui o compilador a fazer, crie uma função simples que usa o modificador _saveregs, e, depois, gere uma listagem em linguagem assembly do código-fonte.

738 INTRODUÇÃO ÀS LISTAS DINÂMICAS

Na seção Matrizes, Ponteiros e Estruturas, anteriormente, você aprendeu como agrupar informações relacionadas em uma única variável. Se seu programa precisa trabalhar com um número fixo de ocorrências de estrutura, ele pode criar uma matriz de estruturas. No entanto, à medida que seus programas se tornarem mais complexos, algumas vezes você não saberá de antemão de quantas estruturas precisará. Nesses casos, você tem duas escolhas. Primeiro, seu programa pode alocar memória dinamicamente para a matriz de estruturas. Segundo, seus programas podem criar uma *lista ligada* de estruturas, onde uma estrutura aponta para a próxima estrutura. A Figura 738 ilustra uma lista ligada de nomes de arquivo.

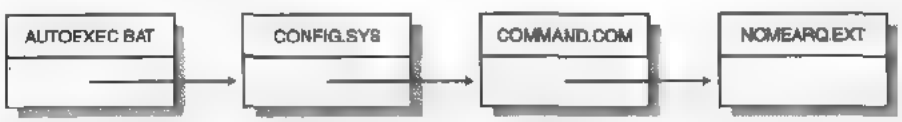

Figura 738 Uma lista ligada de nomes de arquivo.

Em geral, o programa mantém um ponteiro para o início da lista. Um ponteiro para *NULL* indica a última entrada na lista.

739 DECLARANDO UMA ESTRUTURA DE LISTA LIGADA

Para criar uma lista ligada, um dos membros da estrutura precisa ser um ponteiro para uma estrutura do mesmo tipo. Por exemplo, considere a seguinte estrutura:

```
struct ListaArq
 {
   char nomearq[64];
   struct ListaArq *proximo;
 };
```

O membro *nomearq* contém um nome de arquivo. O membro *prox* é um ponteiro para a próxima entrada na lista. Para criar e, mais tarde, percorrer uma lista ligada, seus programas normalmente usarão pelo menos duas

variáveis. A variável *inicio* é uma estrutura. Seu membro *proximo* conterá um ponteiro para o início da lista ou *NULL*, se a lista estiver vazia. A variável *nodo* será um ponteiro para o nodo atual, como mostrado aqui:

```
struct ListaArq inicio, *nodo;
```

CRIANDO UMA LISTA LIGADA 740

Para criar uma lista ligada, seus programas devem seguir os seguintes passos:

1. Declarar a estrutura que defina as entradas da lista.
2. Declarar as variáveis *inicio* e **nodo*.
3. Atribuir a *inicio.proximo* o valor *NULL* para indicar uma lista vazia.

Para cada entrada na lista, seus programas devem executar os seguintes passos:

1. Encontrar o final da lista para que *nodo->proximo* seja *NULL*.
2. Alocar memória para a nova entrada e atribuir o valor da posição inicial de memória ao membro ponteiro *nodo >proximo*.
3. Atribuir a nodo o valor de *nodo->proximo*.
4. Atribuir os valores do membro a nodo.
5. Atribuir a *nodo->proximo* o valor *NULL* para indicar que ele é o novo final da lista.

EXEMPLO DE UMA LISTA LIGADA SIMPLES 741

Na dica anterior você aprendeu os passos que seus programas precisam seguir para criar uma lista ligada. O programa a seguir, *I_10list.c*, cria uma lista ligada cujas entradas contêm os números de 1 a 10:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   int i;
   struct ListaEnt
   {
     int numero;
     struct ListaEnt *proximo;
   } inicio, *nodo;

   inicio.proximo = NULL;  // Lista vazia
   nodo = &inicio;         // Aponta para o início da lista
   for (i = 1; i <= 10; i++)
     {
       nodo->proximo = (struct ListaEnt *) malloc(sizeof(struct ListaEnt));
       nodo = nodo->proximo;
       nodo->numero = i;
       nodo->proximo = NULL;
     }
    // Exibe a lista
    nodo = inicio.proximo;
    while (nodo)
      {
        printf("%d ", nodo->numero);
        nodo = nodo->proximo;
      }
 }
```

742 COMPREENDENDO COMO PERCORRER UMA LISTA LIGADA

Na dica anterior você escreveu o programa *1_10list.c*, que criou uma lista ligada simples com entradas que contêm os números de 1 a 10. O programa usa o laço a seguir para exibir os elementos na lista:

```
// Exibe a lista
nodo = inicio.proximo;
while (nodo)
  {
    printf("%d ", nodo->numero);
    nodo = nodo->proximo;
  }
```

Dentro desse fragmento de código, a variável *inicio.proximo* aponta para o primeiro elemento na lista. Como você pode ver, o código atribui o endereço do primeiro elemento a *nodo*. Da mesma forma, como você se lembrará, *NULL* indica o final da lista. Portanto, o laço simplesmente testa o valor atual de nodo para ver se ele é *NULL*. Se *nodo* não for *NULL*, o laço exibirá o valor do elemento e atribuirá a *nodo* o endereço do próximo elemento na lista.

743 CRIANDO UMA LISTA MAIS ÚTIL

Na Dica 741 você criou uma lista ligada simples que continha os números de 1 a 10. O programa a seguir, *listarq.c*, cria uma lista ligada que contém todos os nomes de arquivo no diretório atual:

```
#include <stdio.h>
#include <dirent.h>
#include <alloc.h>
#include <string.h>

void main(int argc, char *argv[])
 {
   DIR *pont_diretorio;
   struct dirent *entrada;
   struct ListaArq
   {
     char nomearq[64];
     struct ListaArq *proximo;
   } inicio, *nodo;

   if ((pont_diretorio = opendir(argv[1])) == NULL)
     printf("Erro ao abrir %s\n", argv[1]);
   else
     {
        inicio.proximo = NULL;
        nodo = &inicio;
        while (entrada = readdir(pont_diretorio))
         {
            nodo->proximo = (struct ListaArq *)
                malloc(sizeof(struct ListaArq));
            nodo = nodo->proximo;
            strcpy(nodo->nomearq, entrada);
            nodo->proximo = NULL;
         }
        closedir(pont_diretorio);
        nodo = inicio.proximo;
        while (nodo)
         {
            printf("%s\n", nodo->nomearq);
            nodo = nodo->proximo;

         }
```

```
    }
}
```

Listarq.c usa a função *readdir* para ler as entradas do diretório. *Listarq.c* então alocará memória para conter a entrada e copiar o nome de arquivo correspondente para a entrada na lista. Após o programa acrescentar todos os arquivos à lista, ele percorrerá a lista em um laço, e exibirá cada entrada.

Anexando um Elemento na Lista 744

Cada programa de lista ligada que este livro apresentou até aqui criou a lista ligada inteira de uma só vez, normalmente dentro de um laço *while* ou *for*. Dependendo do seu programa, você provavelmente em algum ponto desejará acrescentar elementos à lista em diferentes ocasiões. O modo mais fácil de acrescentar um elemento é colocá-lo no final. Para incluir um item no final de uma lista ligada, você percorrerá a lista até encontrar o elemento que contém o membro *proximo* que aponta para *NULL*, como mostrado aqui:

```
nodo = &inicio;
while (nodo->proximo)
   nodo = nodo->proximo;
```

Quando *nodo->proximo* aponta para *NULL*, você encontrará o final da lista e poderá, portanto, alocar memória para o novo elemento, como mostrado aqui:

```
nodo ->proximo = malloc(tamanho_necessario);
```

Em seguida, atribua o valor que quiser ao item (dentro do item *membro*) e faça o campo *proximo* do novo item apontar para NULL, como mostrado aqui:

```
nodo = nodo->proximo;
nodo->membro = algum_valor;
nodo->proximo = NULL;
```

Em alguns casos, você pode querer que seus programas coloquem elementos em uma posição específica na lista. Você aprenderá a posicionar os elementos na dica a seguir.

Inserindo um Elemento na Lista 745

Na dica anterior você aprendeu como anexar itens a uma lista ligada. Dependendo da função do seu programa, algumas vezes você desejará colocar itens em posições específicas em uma lista. Se você quiser criar uma lista ligada que contenha os nomes classificados de arquivos no diretório atual, então seu programa precisará colocar cada nome de arquivo na lista na posição correta. Para inserir um item em uma posição específica em uma lista, seus programas normalmente controlarão o *nodo* inicial, o *nodo* atual e o *nodo* anterior. Quando seu programa precisar inserir um novo elemento, ele efetuará o seguinte processamento:

```
struct ListaMembro inicio, *nodo, *anterior, *novo;

// Código que efetua inserção de um elemento entre
// os elementos apontados por nodo e anterior
novo = malloc(sizeof(struct ListaMembro);
novo->proximo = nodo;
anterior->proximo = novo;
novo->membro = algum_valor;
```

Exibindo um Diretório Classificado 746

Na dica anterior você viu que, para inserir um elemento em uma lista ligada simples (onde cada elemento contém um ponteiro para o próximo elemento), seus programas precisam controlar os nodos atual e anterior (elementos da lista). O programa a seguir, *classlist.c*, insere elementos em uma lista para criar uma lista que contém os nomes de arquivo classificados do diretório atual:

```
#include <stdio.h>
#include <dirent.h>
#include <alloc.h>
```

```
#include <string.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   DIR *pont_diretorio;
   struct dirent *entrada;
   struct ListaArq
   {
     char nomearq[64];
     struct ListaArq *proximo;
   } inicio, *nodo, *anterior, *novo;

   if ((pont_diretorio = opendir(argv[1])) == NULL)
     printf("Erro ao abrir %s\n", argv[1]);
   else
    {
      inicio.proximo = NULL;
      while (entrada = readdir(pont_diretorio))
       {
         // Encontra a posição correta
         anterior = &inicio;
         nodo = inicio.proximo;
        while ((nodo) && (strcmp(entrada, nodo->nomearq) > 0))
         {
            nodo = nodo->proximo;
            anterior = anterior->proximo;
         }
         novo = (struct ListaArq *)
                malloc(sizeof(struct ListaArq));
         if (novo == NULL)
         {
            printf("Memória insuficiente para armazenar a lista\n");
            exit(1);
         }
         novo->proximo = nodo;
         anterior->proximo = novo;
         strcpy(novo->nomearq, entrada);
      }
      closedir(pont_diretorio);
      nodo = inicio.proximo;
      while (nodo)
      {
         printf("%s\n", nodo->nomearq);
         nodo = nodo->proximo;
      }
   }
 }
```

747 EXCLUINDO UM ELEMENTO DE UMA LISTA

Na Dica 745 você aprendeu como inserir itens em uma lista ligada. À medida que seus programas manipularem listas ligadas, algumas vezes você precisará excluir um elemento da lista. Remover um elemento de uma lista ligada é muito similar a uma operação de inserção, na qual você precisa manipular os ponteiros para os nodos atual e anterior. Após seu programa localizar o primeiro elemento que ele quer excluir, ele poderá usar código similar ao seguinte para remover o nodo:

```
anterior->proximo = nodo->proximo;
free(nodo);
```

O programa a seguir, *remove5.c*, cria uma lista ligada que contém os números de 1 até 10. O programa então pesquisa na lista o elemento que contém o número 5 e remove esse elemento:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   int i;
   struct ListaEnt
   {
     int numero;
     struct ListaEnt *proximo;
   } inicio, *nodo, *anterior;

   inicio.proximo = NULL;  // Lista vazia
   nodo = &inicio;         // Aponta para o início da lista
   for (i = 1; i <= 10; i++)
    {
       nodo->proximo = (struct ListaEnt *)
              malloc(sizeof(struct ListaEnt));
       nodo = nodo->proximo;
       nodo->numero = i;
       nodo->proximo = NULL;
    }
   nodo = inicio.proximo;     // Remove o número 5
   anterior = &inicio;
   while (nodo)
     if (nodo->numero == 5)
       {
         anterior->proximo = nodo->proximo;
         free(nodo);
         break;        // Encerra o laço
       }
     else
       {
         nodo = nodo->proximo;
         anterior = anterior->proximo;
       }
   nodo = inicio.proximo;     // Exibe a lista
   while (nodo)
     {
       printf("%d ", nodo->numero);
       nodo = nodo->proximo;
     }
 }
```

USANDO UMA LISTA DUPLAMENTE LIGADA 748

Uma lista ligada simples tem esse nome porque cada elemento contém um ponteiro para o próximo elemento. Você aprendeu que, para inserir elementos em uma lista ligada simples, seus programas precisam manter ponteiros para o elemento atual e para o anterior. Para simplificar o processo de inserir e remover os elementos da lista, seus programas podem usar uma *lista duplamente ligada*. Em uma lista duplamente ligada, cada elemento mantém um ponteiro para o elemento anterior e para o próximo na lista. A Figura 748 ilustra uma lista duplamente ligada.

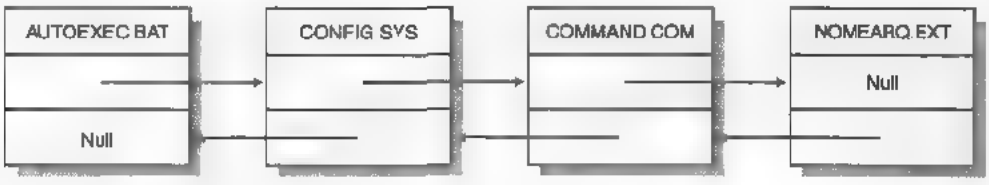

Figura 748 Uma lista duplamente ligada mantém dois ponteiros.

A estrutura a seguir ilustra uma estrutura de lista duplamente ligada:

```
struct ListaArq
 {
   char nomearq[64];
   struct ListaArq *proximo;
   struct ListaArq *anterior;
 };
```

Quando seus programas usam uma lista duplamente ligada, o programa pode percorrer a lista da esquerda para a direita ou da direita para a esquerda. Portanto, a lista precisa manter dois ponteiros *NULL*. Quando seu programa percorre a lista da esquerda para a direita, ele deverá saber quando chegou ao final da lista porque *nodo->proximo* é NULL. Da mesma forma, quando seu programa percorre a lista da direita para a esquerda, ele deverá saber que chegou ao final da lista quando *nodo->anterior* for *NULL*, o que indica o final da lista.

749 CRIANDO UMA LISTA DUPLAMENTE LIGADA SIMPLES

Na dica anterior você aprendeu que uma lista duplamente ligada simplifica o processo de inserir e remover elementos da lista. O programa a seguir, *dbl_1_10.c*, usa uma lista duplamente ligada para exibir os números de 1 a 10 do início para o fim e do fim para o começo:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   int i;
   struct ListaEnt
   {
     int numero;
     struct ListaEnt *proximo;
     struct ListaEnt *anterior;
   } inicio, *nodo;

   inicio.proximo = NULL;  // Lista vazia
   inicio.anterior = NULL;
   nodo = &inicio;         // Aponta para o inicio da lista
   for (i = 1; i <= 10; i++)
    {
      nodo->proximo = (struct ListaEnt *)
            malloc(sizeof(struct ListaEnt));
      nodo->proximo->anterior = nodo;
      nodo = nodo->proximo;
      nodo->numero = i;
      nodo->proximo = NULL;
    }
    nodo = inicio.proximo;     // Exibe a lista
    do
    {
        printf("%d ", nodo->numero);
        nodo = nodo->proximo;
    } while(nodo->proximo); // Mostra o 10 somente uma vez
    do
    {
        printf("%d ", nodo->numero);
        nodo = nodo->anterior;
    } while (nodo->anterior);
 }
```

COMPREENDENDO NODO->ANTERIOR->PRÓXIMO 750

Como você aprendeu, trabalhar com listas duplamente ligadas simplifica as operações de inserção e de exclusão de elemento. À medida que você for examinando os programas que trabalham com listas duplamente ligadas, você poderá encontrar comandos tais como os seguintes:

```
nodo->anterior->proximo = novo_nodo;
```

À medida que você examina esses comandos, trabalhe da esquerda para a direita. A Figura 750 ilustra como o compilador C resolve os ponteiros.

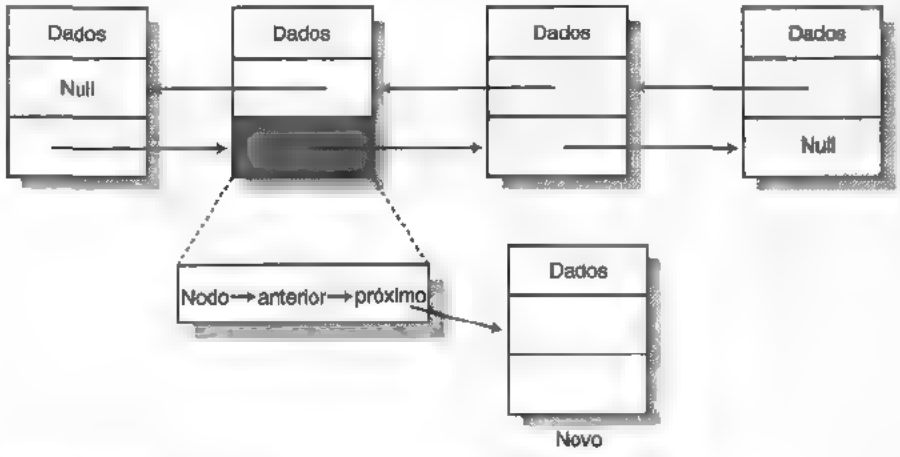

Figura 750 Resolvendo uma complexa operação de ponteiro.

REMOVENDO UM ELEMENTO DE UMA LISTA DUPLAMENTE LIGADA 751

Uma lista duplamente ligada simplifica o processo de inserir e remover elementos da lista. O programa a seguir, *remove_7.c*, cria uma lista duplamente ligada que contém os números de 1 a 10. O programa então pesquisa na lista o elemento que contém o número 7 e o remove:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   int i, encontrado;
   struct ListaEnt {
     int numero;
     struct ListaEnt *proximo;
     struct ListaEnt *anterior;
   } inicio, *nodo;

   inicio.proximo = NULL;  // Lista vazia
   inicio.anterior = NULL;
   nodo = &inicio;         // Aponta para o início da lista
   for (i = 1; i <= 10; i++)
    {
       nodo->proximo = (struct ListaEnt *)
                       malloc(sizeof(struct ListaEnt));
       nodo->proximo->anterior = nodo;
       nodo = nodo->proximo;
       nodo->numero = i;
       nodo->proximo = NULL;
```

```
    }
    // Remove o elemento
    nodo = inicio.proximo;
    encontrado = 0;
    do
    {
      if (nodo->numero == 7)
        {
          encontrado = 1;
          nodo->anterior->proximo = nodo->proximo;
          nodo->proximo->anterior = nodo->anterior;
          free(nodo);
        }
      else
        nodo = nodo->proximo;
    } while ((nodo) && (! encontrado)); // Exibe 10 somente uma vez
    nodo = inicio.proximo;
    do {
        printf("%d ", nodo->numero);
        nodo = nodo->proximo;
    } while (nodo);
}
```

752 INSERINDO UM ELEMENTO EM UMA LISTA DUPLAMENTE LIGADA

Como você aprendeu, as listas duplamente ligadas simplificam a inserção e a exclusão de elementos da lista. O programa a seguir, *cria1_10.c*, cria uma lista que contém os números 1, 3, 5, 7 e 9. O programa então insere os números 2, 4, 6, 8 e 10 nas posições corretas dentro da lista:

```
#include <stdio.h>
#include <alloc.h>

void main(void)
 {
   int i;
   struct ListaEnt {
     int numero;
     struct ListaEnt *proximo;
     struct ListaEnt *anterior;
   } inicio, *nodo, *novo;

   inicio.proximo = NULL;  // Lista vazia
   inicio.anterior = NULL;
   nodo = &inicio;         // Aponta para o início da lista
   for (i = 1; i < 10; i += 2)
   {
      nodo->proximo = (struct ListaEnt *)
             malloc(sizeof(struct ListaEnt));
      nodo->proximo->anterior = nodo;
      nodo = nodo->proximo;
      nodo->numero = i;
      nodo->proximo = NULL;
   }

   for (i = 2; i <= 10; i += 2)
   {
     int encontrado = 0;
```

```
    novo = (struct ListaEnt *) malloc(sizeof (struct ListaEnt));
    novo->numero = i;
    nodo = inicio.proximo;
    do {
      if (nodo->numero > novo->numero)
      {
         novo->proximo = nodo;
         novo->anterior = nodo->anterior;
         nodo->anterior->proximo = novo;
         nodo->anterior = novo;
         encontrado = 1;
      }
      else
        nodo = nodo->proximo;
    } while ((nodo->proximo) && (! encontrado));
    if (! encontrado)
      if (nodo->numero > novo->numero)
      {
         novo->proximo = nodo;
         novo->anterior = nodo->anterior;
         nodo->anterior->proximo = novo;
         nodo->anterior = novo;
      }
      else
       {
          novo->proximo = NULL;
          novo->anterior = nodo;
          nodo->proximo = novo;
       }
  }
  // Exibe a lista
  nodo = inicio.proximo;
  do {
     printf("%d ", nodo->numero);
     nodo = nodo->proximo;
  } while (nodo);
}
```

COMPREENDENDO OS PROCESSOS-FILHO 753

Quando você roda um programa, esse programa pode rodar um segundo programa, chamado *processo-filho*. O programa que roda esse segundo programa é chamado *pai*. Dependendo das suas necessidades, o processo-filho pode rodar até o final e o pai pode continuar, ou o filho pode tomar o lugar do pai, sobrescrevendo-o na memória. Quando o programa-filho roda até o final e o pai continua, a execução do filho é chamada *geração*. Quando o processo-filho substitui o pai na memória, o programa precisa *exec* o filho. Para ajudar seus programas a efetuar esse processamento, a biblioteca de execução de C fornece dois tipos de funções de biblioteca de execução: *spawn* e *exec*. As Dicas de 754 e 757 discutem detalhadamente essas rotinas de execução.

GERANDO UM PROCESSO-FILHO 754

Como você aprendeu na dica anterior, quando um programa gera uma tarefa-filho, o programa-pai suspende seu processamento enquanto o processo-filho roda, e, depois, mais tarde, continua. Para gerar um processo-filho, seus programas podem usar a função *spawnl*, como mostrado aqui:

```
#include <process.h>
#include <stdio.h>
int spawnl(int modo, char *filho, char *arg0, ... ,char *argn, NULL);
```

O parâmetro *filho* é um ponteiro para uma string de caracteres que especifica o nome do arquivo executável que contém o processo-filho. Os parâmetros *arg0* até *argn* especificam os argumentos da linha de comando do processo-filho. O parâmetro *modo* especifica como seu programa roda o processo-filho. A Tabela 754.1 lista os possíveis valores de modos.

Tabela 754.1 Modos de execução do processo-filho

Valor	Modo de Execução
P_NOWAIT	*O processo-pai continua a executar em paralelo com o filho (não disponível para* os programas baseados no DOS)
P_OVERLAY	O processo-filho sobrescreve o pai na memória
P_WAIT	O processo pai reinicia após o filho terminar

Se a função *spawnl* for bem-sucedida, ela retornará o valor 0 Se ocorrer um erro, a função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 754.2.

Tabela 754.2 Os valores de erro que ***spawnl*** retorna.

Valor	Descrição
E2BIG	A lista de argumentos é longa demais
EINVAL	Argumento inválido
ENOENT	O programa-filho não foi encontrado
ENOEXEC	Erro no formato
ENOMEM	Memória insuficiente

Para compreender melhor os processos-filho, crie o programa *filho.c*, que exibirá seus argumentos na linha de comando e as variáveis do ambiente, como mostrado aqui:

```
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[], char *env[])
 {
   printf("Linha de comando\n");
   while (*argv)
       puts(*argv++);
   printf("Variáveis do ambiente\n");
   while (*env)
       puts(*env++);
 }
```

Compile o programa. Em seguida, crie o programa *spawnl.c*, que usa a função *spawnl* para executar o processo-filho:

```
#include <process.h>
#include <stdio.h>

void main(void)
 {
   printf("Prestes a chamar o processo-filho\n\n");
   spawnl(P_WAIT, "FILHO.EXE", "FILHO.EXE", "AAA", "BBB", "CCC", NULL);
   printf("\n\nVoltando do processo-filho\n");
 }
```

Quando você executar o programa *spawnl*, sua tela exibirá uma mensagem dizendo que está prestes a chamar o processo-filho. Em seguida, o processo-filho rodará, exibindo seus argumentos da linha de comando e as variáveis do ambiente. Após o processo-filho terminar, o programa exibirá uma mensagem dizendo que retornou do processo-filho.

Usando Outras Funções spawnlxx 755

Na dica anterior você aprendeu que a função *spawnl* lhe permite rodar um processo-filho. Se você examinar a biblioteca de execução de C, encontrará várias outras funções *spawnlxx*, como mostrado aqui:

```
#include <stdio.h>
#include <process.h>

int spawnle(int modo, char *filho, char *arg0, ..., char *argn, NULL, char
          *ambiente);
int spawnlp(int modo, char *filho, char *arg0, ..., char *argn, NULL);
int spawnlpe(int modo, char *filho, char *arg0, ..., char *argn, NULL, char
          *ambiente);
```

Se qualquer função *spawnlxx* for bem-sucedida, ela retornará o valor 0. Se ocorrer um erro, cada função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 755.

Tabela 755 Valores de erro que as funções *spawnlxx* retornam.

Valor	Descrição
E2BIG	A lista de argumentos é grande demais
EINVAL	Argumento inválido
ENOENT	Programa-filho não-encontrado
ENOEXEC	Erro no formato
ENOMEM	Memória insuficiente

Os parâmetros para as funções *spawnlxx* são similares àquelas que *spawnl* usa, como detalhado na Dica 754. No entanto, as funções *spawnlxx* também usam o parâmetro ambiente, que contém um ponteiro para as variáveis do ambiente do filho. A diferença entre *spawnl* e *spawnlp* é que *spawnlp* e *spawnlpe* pesquisarão o caminho de comando para o processo-filho. O programa a seguir, *spawnlxx.c*, ilustra o uso das funções *spawnlxx*:

```
#include <process.h>
#include <stdio.h>

void main(void)
 {
   char *ambiente[] = { "ARQUIVO=SPAWNLXX.C", "LINGUAGEM=C",
                        "SO=DOS", NULL };
   spawnle(P_WAIT, "FILHO.EXE", "FILHO.EXE",
           "Usando spawnle", "BBB", NULL, ambiente);
   spawnlp(P_WAIT, "FILHO.EXE", "FILHO.EXE",
           "Usando spawnlp", "BBB", NULL);
   spawnlpe(P_WAIT, "FILHO.EXE", "FILHO.EXE",
           "Usando spawnlpe", "BBB", NULL, ambiente);
 }
```

Usando as Funções spawnvxx 756

Na Dica 754 você aprendeu como usar a função *spawnl* para criar um processo-filho. Da mesma forma, na Dica 755 você usou as diferentes funções *spawnlxx*, que lhe permitem passar uma matriz de variáveis do ambiente para o processo-filho. As funções *spawnlxx* também lhe permitem usar o caminho de comando para localizar o processo-filho. Quando você usa as funções *spawnl*, passa os argumentos da linha de comando, como uma lista de parâmetros terminados por *NULL*. Além das funções *spawnlxx*, C fornece uma coleção de funções *spawnvxx* que

lhe permitem passar os parâmetros da linha de comando como uma matriz de strings de caracteres, como mostrado aqui.

```
#include <stdio.h>
#include <process.h>

int spawnv(int modo, char *filho, char *argv[]);
int spawnve(int modo, char *filho, char *argv[], char *env[]);
int spawnvp(int modo, char *filho, char *argv[]);
int spawnvpe(int modo, char *filho, char *argv[], char *env[]);
```

Se qualquer função *spawnvxx* for bem-sucedida, a função retornará o valor 0. Se ocorrer um erro, cada função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 756.

Tabela 756 valores de erro que *spawnvxx* retorna.

Valor	Descrição
E2BIG	A lista de argumento é longa demais
EINVAL	Argumento inválido
ENOENT	Programa-filho não-encontrado
ENOEXEC	Erro no formato
ENOMEM	Memória insuficiente

Os parâmetros para as funções *spawnvxx* são similares àquelas que seu programa passa para as funções *spawnlxx*, exceto que as funções *spawnvxx* passam os argumentos da linha de comando como uma matriz de strings de caracteres. O programa a seguir, *spawnvxx.c*, ilustra as funções *spawnvxx*:

```
#include <stdio.h>
#include <process.h>

void main(void)
 {
   char *ambiente[] = { "NOMEARQ=SPAWNVXX.C", "SO=DOS",
                        "ROTINAS=SPAWNVXX", NULL };
   char *argv[] = { "FILHO.EXE", "AAA", "BBB", NULL };

   spawnv(P_WAIT, "FILHO.EXE", argv);
   spawnve(P_WAIT, "FILHO.EXE", argv, ambiente);
   spawnvp(P_WAIT, "FILHO.EXE", argv);
   spawnvpe(P_WAIT, "FILHO.EXE", argv, ambiente);
 }
```

757 EXECUTANDO UM PROCESSO-FILHO

Como você aprendeu na Dica 753, quando um programa executa uma tarefa-filha, o processo-filho sobrescreve o programa-pai na memória. Como o filho sobrescreve o pai, o processo-pai nunca reinicia. Para executar um processo-filho, seus programas podem usar a função *execl*, como mostrado aqui:

```
#include <process.h>
#include <stdio.h>

int execl(char *filho, char *arg0,...,char *argn, NULL);
```

O parâmetro *filho* é um ponteiro para uma string de caracteres que especifica o nome do arquivo executável que contém o processo-filho. Os parâmetros *arg0* até *argn* especificam os argumentos da linha de comando do processo-filho.

Se a função *execl* for bem-sucedida, ela não retornará um valor. Se um erro ocorrer, a função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 757.

Tabela 757 Os valores de erro que *execl* retorna.

Valor	Descrição
E2BIG	A lista de argumentos é longa demais
EINVAL	Argumento inválido
ENOENT	Programa-filho não encontrado
ENOEXEC	Erro de formato
ENOMEM	Memória insuficiente

Para compreender melhor a execução dos processos-filhos, crie o seguinte programa, *filho.c*, que exibirá seus argumentos da linha de comando e as variáveis do ambiente.

```
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[], char *ambiente[])
 {
   printf("Linha de comando\n");
   while (*argv)
      puts(*argv++);
   printf("Variáveis do ambiente\n");
   while (*ambiente)
      puts(*ambiente++);
 }
```

Compile o programa. Em seguida, crie o seguinte programa, *execl.c*, que usa a função *execl* para executar o processo-filho.

```
#include <stdio.h>
#include <process.h>

void main(void)
 {
   printf("Prestes a chamar o processo-filho\n\n");
   execl("FILHO.EXE", "FILHO.EXE", "AAA", "BBB", "CCC", NULL);
   printf("\n\Voltando do processo-filho -- NÃO DEVE APARECER\n");
 }
```

Quando você executar o programa *execl*, sua tela exibirá uma mensagem dizendo que está prestes a chamar o processo-filho. Em seguida, o processo-filho rodará, exibindo seus argumentos da linha de comando e as variáveis do ambiente. O processo-filho sobrescreve o processo-pai, de modo que, após o processo-filho terminar, nenhum processamento adicional ocorre.

Usando Outras Funções execlxx 758

Na dica anterior você aprendeu que a função *execl* lhe permite rodar um processo-filho. Se você examinar a biblioteca de execução de C, encontrará várias outras funções *execlxx*, como mostrado aqui:

```
#include <process.h>
#include <stdio.h>

int execle(char *filho, char *arg0, ...,char *argn, NULL, char *ambiente);
int execlp(char *filho, char *arg0, ...,char *argn, NULL);
int execlpe(char *filho, char *arg0, ...,char *argn, NULL, char *ambiente);
```

Se qualquer função *execlxx* for bem-sucedida, a função não retornará um valor. Se um erro ocorrer, a função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 758.

Tabela 758 Valores de erro que a função *execlxx* retorna.

Valor	Descrição
E2BIG	A lista de argumento é muito longa
EINVAL	Argumento inválido
ENOENT	Programa-filho não-encontrado
ENOEXEC	Erro de formato
ENOMEM	Memória insuficiente

Os parâmetros para essas funções são similares àqueles que *spawnl* usa, com a exceção do parâmetro *ambiente*, que contém um ponteiro para as entradas do ambiente do filho. (A Dica 754 descreve *spawnl* em detalhe.) A diferença entre *execl* e *execlp* é que as funções que contêm a letra *p* pesquisarão o processo-filho no caminho de comandos. O programa a seguir, *execlpe.c*, ilustra o uso da função *execlpe*:

```
#include <process.h>
#include <stdio.h>

void main(void)
 {
   char *ambiente[] = { "ARQUIVO=EXECLPE.C", "LINGUAGEM=C", "SO=DOS", NULL };

   execlpe("FILHO.EXE", "FILHO.EXE", "Usando execlpe", "BBB", NULL, ambiente);
 }
```

759 USANDO AS FUNÇÕES EXECVXX

Na Dica 757 você aprendeu como usar a função *execl* para criar um processo-filho. Da mesma forma, na dica anterior, você usou as diferentes funções *execlxx*, que lhe permitem passar uma matriz de variáveis do ambiente para o processo-filho e também lhe permitem usar o caminho de comando para localizar o processo-filho. Ao usar as funções *execl*, você passa os argumentos da linha de comando como uma lista de parâmetros terminados por *NULL*. Além das funções *execlxx*, C fornece uma coleção de funções *execvxx* que lhe permitem passar os parâmetros da linha de comando como uma matriz de strings de caracteres, como mostrado aqui:

```
#include <stdio.h>
#include <process.h>

int execv(char *filho, char *argv[]);
int execve(char *filho, char *argv[], char *env[]);
int execvp(char *filho, char *argv[]);
int execvpe(char *filho, char *argv[], char *env[]);
```

Se qualquer função *execvxx* for bem-sucedida, a função não retornará um valor. Se ocorrer um erro, a função retornará o valor -1, e definirá a variável global *errno* com um dos valores listados na Tabela 759.

Tabela 759 Os valores de erro que a função *execvxx* retorna.

Valor	Descrição
E2BIG	A lista de argumentos é longa demais
EINVAL	Argumento inválido
ENOENT	Programa-filho não-encontrado
ENOEXEC	Erro de formato
ENOMEM	Memória insuficiente

Os parâmetros para as funções *execvxx* são similares àquelas que o programa passa para as funções *execlxx*, exceto que *execvxx* passa os argumentos da linha de comando como uma matriz de strings de caracteres. O programa a seguir, *execvpe.c*, ilustra a função *execvpe*:

```
#include <stdio.h>
#include <process.h>
void main(void)
```

```
{
  char *ambiente[]={ "NOMEARQ=SPAWNVXX.C", "SO=DOS", "ROTINA=EXECVPE", NULL };
  char *argv[]={ "FILHO.EXE", "AAA", "BBB", NULL };

  execvpe("FILHO.EXE", argv, ambiente);
}
```

COMPREENDENDO OS OVERLAYS 760

Como você aprendeu na seção Gerenciamento de Memória no Windows, anteriormente, o DOS restringe os programas aos primeiros 640Kb de memória. Para suportar os programas maiores, os programas mais antigos dividiam seu código em áreas fixas chamadas *overlays*. À medida que o programa executava, ele carregava diferentes seções de overlay, conforme necessário. Embora os overlays permitam que os programadores escrevam e compilem programas muito grandes, os overlays requerem que os programas controlem quais overlays estão carregados no momento, bem como quais overlays contêm as funções desejadas. Como você pode imaginar, esse processamento pode ser difícil, pois requer que os programas aplicativos forneçam as operações de gerenciamento de memória que um sistema operacional normalmente fornece.

Para ajudar seus programas a carregar e executar os overlays, o DOS fornece um serviço do sistema que carrega um arquivo de overlay, e, depois, transfere o controle para o início do arquivo. Para detalhes específicos sobre o uso do DOS para carregar e executar overlays, consulte o livro *DOS Programming: The Complete Reference*, Osborne/McGraw Hill, 1991, de Kris Jamsa. Talvez você ache difícil usar o DOS para gerenciar os overlays. No entanto, muitos compiladores fornecem ferramentas de gerenciamento de overlays que seus programas podem usar. Para maiores informações sobre os gerenciadores de memória, consulte a documentação do seu compilador. O *Turbo C++ Lite* não incluí seu próprio gerenciador de memória.

COMPREENDENDO AS INTERRUPÇÕES 761

Uma *interrupção* é um evento que faz o computador parar temporariamente a tarefa que está executando no momento para poder trabalhar em uma segunda tarefa. Quando o processamento da interrupção terminar, o computador reinicia a tarefa inicial como se a interrupção nunca tivesse ocorrido. O PC suporta interrupções de *hardware* e de *software*. A seção Serviços DOS e BIOS deste livro discute como você pode usar as interrupções de software para acessar as interrupções do DOS e da BIOS. Por outro lado, dispositivos, tais como a unidade de disco ou o relógio do sistema do PC, geram interrupções de hardware. Uma *rotina de tratamento de interrupção* é um software que responde a uma interrupção específica. Normalmente, os programadores experientes escrevem rotinas de tratamento de interrupção em linguagem assembly. No entanto, os novos compiladores permitem que você escreva essas rotinas em C.

Os primeiros 1.024 bytes da memória do PC contêm os endereços de segmento e de deslocamento (chamados *vetores de interrupção*) para as 256 interrupções do PC. Quando uma interrupção específica ocorre, o PC coloca na pilha o ponteiro da instrução (IP), o segmento de código e o registrador de flags (o estado da máquina). O PC então encontra o endereço da rotina de tratamento de interrupção usando o vetor de interrupção. A rotina de tratamento de interrupção então salva os registradores do PC e inicia seu processamento. Após a rotina de tratamento de interrupção terminar, ela retira os registradores da pilha e executa uma instrução IRET, que restaura o registrador de flags, o CS e o IP. Os comandos em linguagem assembly a seguir, por exemplo, ilustram o layout típico de uma rotina de tratamento de interrupção:

```
; Salva os registradores na pilha
PUSH      AX
PUSH      BX
PUSH      CX
PUSH      DX
PUSH      SI
PUSH      DI
PUSH      DS
PUSH      ES

;  Executa as instruções de tratamento de interrupção

;  Restaura os registradores da pilha
POP       ES
```

```
POP     DS
POP     DI
POP     SI
POP     DX
POP     CX
POP     BX
POP     AX

;  Retorna à tarefa que estava executando antes
IRET
```

Quando você define sua própria rotina de tratamento de interrupção com linguagem assembly, atualiza o vetor de interrupção para que ele aponte para a rotina de interrupção que você criou. Quando seu programa terminar, você precisará restaurar a definição original do vetor de interrupção.

762 AS INTERRUPÇÕES DO PC

Dentro do PC, os primeiros 1.024 bytes da memória do computador contêm os endereços (vetores) das 256 interrupções do PC. O sistema operacional não usa muitas interrupções, o que as deixa disponíveis para os propósitos dos seus programas. A Tabela 762 lista os vetores de interrupção do PC e seus usos.

Tabela 762 Os vetores de interrupção do pc e seus usos.

Interrupção	Propósito	Interrupção	Propósito
00H	Hardware — divisão por 0	1EH	Tabela de parâmetros do disco
01H	Execuçao passo a passo — hardware	1FH	Tabela de caracteres gráficos
02H.	Interrupção não-mascarável	20H	Terminar programa — DOS
03H	Definição de breakpoint do depurador	21H	Serviços do sistema do DOS
04H	Extravasamento aritmético	22H	Terminar programa
05H	Serviço print screen da BIOS	23H	Ctrl+Break — DOS
08H	Pulso do relógio — IRQ0	24H	Erro crítico do DOS
09H	IRQ1 — teclado	25H	Leitura do disco — DOS
0AH	IRQ2	26H	Gravação no disco — DOS
0BH	IRQ3 — COM2	27H	Terminar programa residente — DOS
OCH	IRQ4 — COM2	28H	DOS ocioso
0DH	IRQ5 PC/AT LPT1	29H	Função *putchar* rápida do DOS
0EH	IRQ6 — Disquete	2AH	Serviços do MS-Net
0FH	IRQ7 LPT1	2EH	Carregador principal do DOS
10H	Serviços de vídeo da BIOS	2FH	Multiplex do MS-DOS
11H	Lista do equipamento da BIOS	33H	Serviços do mouse
12H	Tamanho da memória da BIOS	40H	Vetor do disquete
13H	Serviços de disco da BIOS	41H	Tabela de parâmetros do disco rígido
14H	Serviços de comunicação da BIOS	42H	Redirecionamento da BIOS para EGA
15H	Serviços diversos da BIOS	43H	Tabela de parâmetros de EGA
16H	Serviços de teclado da BIOS	44H	Tabela de caracteres de EGA
17H	Serviços de impressão da BIOS	4AH	Alarme da Int 70H do PC/AT
18H	Chama o BASIC em ROM	5CH	Serviços do NetBIOS
19H	Reinicialização do sistema	67H	Serviços de EMS
1AH	Hora do dia mantida pela BIOS	70H	IRQ8 — Tempo real do PC/AT
1BH	Tratamento de Ctrl-Break	71H	IRQ9 — Redirecionamento da INT 0AH do PC
1CH	Chamada pela rotina da INT 8	75H	IRQ13 — co-processador matemático do PC/AT
1DH	Tabela de parâmetros do vídeo		

USANDO A PALAVRA-CHAVE INTERRUPT 763

Como você aprendeu, o DOS lhe permite criar suas próprias rotinas de tratamento de interrupção. Se você estiver usando o *Turbo C++ Lite*, a palavra-chave *interrupt* facilitará a criação de uma rotina de tratamento de interrupção, como mostrado aqui:

```
void interrupt rotina_personalizada()
 {
   // Comandos da rotina
 }
```

Quando o compilador encontra a palavra-chave *interrupt*, ele insere comandos para salvar e restaurar os registradores, como é necessário, e, depois, retorna da rotina usando uma instrução IRET da linguagem assembly. Para compreender o processamento que a palavra-chave *interrupt* efetua, crie um programa que contenha a função *rotina_personalizada* na forma de linguagem assembly, caso seu compilador permita esse tipo de compilação. Se você, mais tarde, examinar o arquivo-fonte em linguagem assembly, verá instruções em linguagem de máquina similares às mostradas na Dica 761.

DETERMINANDO O VETOR DE UMA INTERRUPÇÃO 764

Como você viu um *vetor de interrupção* é o endereço de segmento e de deslocamento do código que trata a interrupção. Para ajudar seus programas a determinar um vetor de interrupção, muitos compiladores baseados no DOS fornecem a função *_dos_getvect*, como mostrado aqui:

```
#include <dos.h>

void interrupt(*_dos_getvect(unsigned num_interrup))();
```

O parâmetro *num_interrup* especifica a interrupção que você quer (de 0 a 255). A função retorna um ponteiro para uma rotina de tratamento de interrupção. O programa a seguir, *vetores.c*, exibe os vetores para todas as interrupções do PC:

```
#include <stdio.h>
#include <dos.h>

void main(void)
 {
   int i;

   for (i = 0; i <= 255; i++)
     printf("Interrupção: %x Vetor: %lx\n", i, _dos_getvect(i));
 }
```

DEFININDO UM VETOR DE INTERRUPÇÃO 765

Quando seus programas criam suas próprias rotinas de tratamento de interrupção, os programas precisam atribuir o vetor de interrupção para apontar para a rotina personalizada de tratamento de interrupção. Para ajudar seus programas a atribuir vetores de interrupção, a maioria dos compiladores baseados no DOS fornece a função *_dos_setvect*, como mostrado aqui:

```
#include <dos.h>
void _dos_setvect(unsigned num_int, void interrupt(*rotina)());
```

O parâmetro *num_int* especifica a interrupção cujo vetor você quer alterar. O parâmetro *rotina* é um ponteiro para a rotina de tratamento da interrupção. A dica a seguir ilustra o uso da função *_dos_setvect*. Quando seu programa criar um vetor de interrupção, ele precisará salvar as definições originais do vetor para que ele possa restaurar o vetor original antes de seu programa terminar. Se um programa terminar sem restaurar o vetor de interrupção, seu sistema poderá se comportar de maneira errática (geralmente, ele poderá deixar de funcionar).

766 HABILITANDO E DESABILITANDO AS INTERRUPÇÕES

Quando seus programas efetuam o tratamento das interrupções, algumas vezes você quer que os programas habilitem e desabilitem as interrupções. Para ajudá-lo a controlar as interrupções, muitos compiladores baseados no DOS fornecem as macros *_disable* e *_enable*, como mostrado aqui:

```
#include <dos.h>

void _disable(void);
void _enable(void);
```

Para funcionar corretamente, o PC precisa gerar interrupções regularmente. Assim sendo, se seus programas desabilitarem as interrupções, eles deverão minimizar o tempo em que as interrupções ficam minimizadas. Comumente, seus programas usarão as macros _disable e _enable quando alterarem um vetor de interrupção com _dos_setvect, como mostrado aqui:

```
_disable();
_dos_setvect(num_int, rotina);
_enable();
```

767 CRIANDO UMA ROTINA SIMPLES DE TRATAMENTO DE INTERRUPÇÃO

Como você aprendeu, criar rotinas de tratamento de interrupção com a palavra-chave *Turbo C++ Lite* é muito mais fácil que criar essas rotinas em assembly. O programa a seguir, *semprtscr.c*, cria uma rotina de tratamento de interrupção que substitui a rotina da BIOS para a tecla print screen, o que imprime o conteúdo da tela quando você pressiona a combinação de teclas Shift+PrtSc. O programa usa a função *_dos_getvect* para determinar a definição original do vetor para que possa restaurá-lá antes de o programa terminar. Pressionar Shift+PrtSc enquanto o programa está ativo invoca sua rotina de tratamento de interrupção, a qual, por sua vez, exibe uma mensagem na tela dizendo que você pressionou a combinação Shift+PrtSc. Quando você pressionar Shift+PrtSc três vezes, o programa *semprtscr.c* terminará:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

int conta = 0;

void interrupt rotina(void)
 {
   conta++;
 }

void main(void)
 {
   void interrupt (*rotina_original)();
   int conta_antigo = 0;

   rotina_original = _dos_getvect(5);
   _disable();  // Desabilita as interrupções durante _dos_setvect
   _dos_setvect(5, rotina);
   _enable();
   printf("Pressione Shift+PrtSc três vezes ou qualquer tecla para
   terminar\n");
   while (conta < 3)
     if (conta != conta_antigo)
       {
```

```
        printf("Shift+PrtSc pressionado\n");
        conta_antigo = conta;
      }

  _disable();
  _dos_setvect(5, rotina_original);
  _enable();
}
```

ENCADEANDO UMA SEGUNDA INTERRUPÇÃO 768

Na dica anterior você aprendeu como escrever uma rotina de interrupção para a operação print screen da BIOS. Dependendo da função que seu programa executar, algumas vezes você poderá querer executar a rotina de tratamento original após sua rotina completar seu processamento. Nesses casos, seu programa poderá usar a função *_chain_interrupt*, como mostrado aqui:

```
#include <dos.h>

void _chain_interrupt(void (interrupt far *rotina)());
```

Por exemplo, o programa a seguir, *contados.c*, mantém um contador do número de vezes que seu programa chama as interrupções específicas do DOS (isto é, *contados.c*, examina o registrador AH para a INT 21H). Após o programa *contados.c* terminar, ele exibirá um contador do número de serviços que seu programa chamou:

```
#include <stdio.h>
#include <dos.h>
#include <dir.h>

int funcao[255];  // Serviços do DOS
void interrupt far (*rotina_original)();

void interrupt far rotina(void)
 {
   char i;

   asm { mov i, ah }
   funcao[i]++;
   _chain_intr(rotina_original);
 }

void main(void)
 {
   int i;

   for (i = 0; i < 255; i++)  // Zera os contadores
      funcao[i] = 0;
   rotina_original = _dos_getvect(0x21);
   _disable();
   _dos_setvect(0x21, rotina);
   _enable();
   printf("Isto é uma mensagem\n");
   fprintf(stdout, "Esta é uma segunda mensagem\n");
   printf("O disco atual é %c\n", getdisk() + 'A');
   _disable();
   _dos_setvect(0x21, rotina_original);
   _enable();
   for (i = 0; i <= 255; i++)
     if (funcao[i])
       printf("Funcao %x chamada %d vezes\n", i, funcao[i]);
 }
```

769 GERANDO UMA INTERRUPÇÃO

Como você aprendeu na seção Serviços DOS e BIOS, anteriormente, a biblioteca de execução de C fornece as funções *intdos* e *int86*, que permitem que seus programas acessem os serviços do DOS e da BIOS. À medida que seus programas forem tratando interrupções específicas, algumas vezes você irá querer gerar uma interrupção para testar suas rotinas de tratamento, e outras vezes seus programas precisarão chamar uma interrupção específica. Para gerar interrupções, seus programas podem usar a função *geninterrupt*, como mostrado aqui:

```
#include <dos.h>

void geninterrupt(int interrupcao);
```

O parâmetro *interrupcao* especifica a interrupção que você quer. O programa a seguir, *genintr.c*, chama a interrupção pouco usada 0xFF para notificar o programa de um evento específico:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>

void interrupt far (*rotina_original)();

void interrupt far rotina(void)
 {
   printf("Algum evento acaba de ocorrer\n");
   _disable();
   _dos_setvect(0xFF, rotina_original);
   _enable();
   exit(0);
 }

void main(void)
 {
   int i = 0;

   rotina_original = _dos_getvect(0xFF);
   _disable();
   _dos_setvect(0xFF, rotina);
   _enable();
   while (i++ < 100)
     ;
   geninterrupt(0xFF);
 }
```

Nota: Quando você programar no Windows, pegará mensagens e eventos, em vez de gerar e pegar interrupções. Você aprenderá mais sobre as mensagens e eventos na Dica 1251.

770 INTERCEPTANDO O TEMPORIZADOR DO PC

Muitos dispositivos dentro e fora do PC precisam executar operações em intervalos específicos. Para efetuar essas operações, o PC fornece um circuito integrado temporizador que gera um sinal 18,2 vezes por segundo. Cada vez que o sinal ocorre, o PC gera a interrupção 8, que atualiza o relógio, e a interrupção 1CH, que seus programas podem interceptar. O programa a seguir, *temporiz.c*, intercepta a interrupção 1CH cada vez que a interrupção ocorre:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

int alfanum = 0;
int contador = 0;

void interrupt far rotina(void)
```

```
  {
    if (++contador == 273)  // 15 segundos
     {
        alfanum = !alfanum;  // Inverte
        contador = 0;
     }
  }

void main(void)
  {
    int i;
    void interrupt far (*rotina_original)();

    rotina_original = _dos_getvect(0x1C);
    _disable();
    _dos_setvect(0x1c, rotina);
    _enable();
    while (! kbhit())
      if (alfanum)
        for (i = 'A'; i <= 'Z'; i++)
           printf("%c\n", i);
      else
        for (i = 0; i <= 100; i++)
           printf("%d\n", i);
     _disable();
     _dos_setvect(0x1c, rotina_original);
     _enable();
  }
```

A rotina de interrupção conta o número de ocorrências e inverte o valor da variável global *alfanum* a cada 15 segundos. Se o valor de *alfanum* for 1, o programa exibirá repetidamente as letras do alfabeto. Se o valor de *alfanum* for 0, o programa exibirá os números de 1 a 100.

COMPREENDENDO OS ERROS CRÍTICOS 771

Como provavelmente você sabe, quando você tenta usar uma unidade de disquete que não contém um disco formatado, o DOS exibirá uma mensagem de erro, seguida pelas conhecidas opções:

```
Abortar, Repetir, Falhar?
```

Tais erros são chamados *erros críticos* porque o DOS não pode resolvê-los sem a ajuda do usuário. Quando um erro crítico ocorre, o DOS chama a interrupção 24H. Quando seus programas interceptam a interrupção 24H, você pode efetuar tratamento de erro crítico, e possivelmente exibir uma mensagem de erro mais significativa ou instrutiva para o usuário. Quando o DOS chama a INT 24H, ele coloca informações consideráveis na pilha que descrevem a causa e a origem do erro. Para uma discussão completa do conteúdo da pilha e as operações que seus programas devem executar ao tratar os erros críticos, consulte o livro *DOS Programming: The Complete Reference*, de Kris Jamsa, Osborne/McGraw Hill, 1991. Como você aprenderá na dica a seguir, a maioria dos compiladores baseados no DOS fornece funções da biblioteca de execução que simplificam o tratamento de erro crítico.

TRATAMENTO DE ERRO CRÍTICO EM C 772

Como a dica anterior descreveu, um erro crítico é um erro a partir do qual o DOS não pode continuar sem a intervenção do usuário. Para ajudar os programas do DOS a efetuar o tratamento de erro crítico, a maioria dos compiladores C fornece as seguintes funções da biblioteca de execução:

```
#include <dos.h>

void _harderr(int (*rotina)());
void _hardresume(int ax_registrador);
void _hardreturn(int valor_rotina);
```

A função *_harderr* lhe permite especificar o nome da função que tratará os erros críticos. A função *_hardresume* permite que seus programas retornem um valor de status para o DOS. A função *_hardreturn*, por outro lado, lhe permite retornar um valor (qualquer valor) para seu programa. O valor que *_hardresume* retorna precisa ser um dos listados na Tabela 772.

Tabela 772 Constantes que ***_hardresume*** retorna.

Constante	Significado
_HARDERR_ABORT	Sinalizar o programa atual
_HARDERR_RETRY	Repetir o serviço que causa o erro
_HARDERR_FAIL	Falhar o serviço que está causando o erro
_HARDERR_IGNORE	Ignorar o erro

O programa a seguir, *errosimp.c*, fornece uma rotina de tratamento de erro crítico que exibirá uma mensagem na tela, e, depois, usará a função *_hardresume* para abortar o programa:

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>

void far handler(unsigned erro_dispo, unsigned cdg_erro,
                 unsigned far *cabecalho_dispo)
 {
   cputs("Erro crítico finalizando o programa\n");
   _hardresume(_HARDERR_ABORT);   // Aborta
 }

void main(void)
 {
   FILE *pa;

   _harderr(handler);
   pa = fopen("A:UMARQ.EXT", "r");
   printf('Mensagem do programa...\n");
   fclose(pa);
 }
```

773 UMA ROTINA DE TRATAMENTO DE ERRO CRÍTICO COMPLETA

Na dica anterior você criou uma rotina de tratamento de erro crítico que exibia uma mensagem, e, depois, finalizava o programa que causou o erro. Se você olhar a rotina de tratamento de erro crítico, verá que ela suporta três parâmetros, como mostrado aqui:

```
void far rotina(unsigned dispo_erro, unsigned cdg_erro,
                unsigned far *cabecalho_dispo)
```

Quando o DOS chama a rotina de tratamento de erro crítico, ele coloca informações sobre o erro na pilha. O parâmetro *dispo_erro* contém um valor de erro que o DOS normalmente passaria para uma rotina de tratamento de erro crítico no registrador *AX*. Se o serviço que falhou definiu o bit 7 do parâmetro *dispo_erro*, o erro é um erro de disco. A Tabela 773.1 lista os valores que o serviço que falhou pode atribuir a *dispo_erro*.

Tabela 773.1 Valores de erro que a variável ***dispo_erro*** retorna.

Bit(s)	Valor	Significado
0	0	Erro de leitura
	1	Erro de gravação
1-2	00	Erro do DOS
	01	Erro na FAT
	10	Erro no diretório
	11	Erro em arquivo

Tabela 773.1 Valores de erro que a variável *dispo_erro* retorna. (Continuação)

Bit(s)	Valor	Significado
3	0	Operação de falha não-permitida
	1	Operação de falha permitida
4	0	Operação de repetição não-permitida
	1	Operação de repetição permitida
5	0	Operação ignorar não-permitida
	1	Operação ignorar permitida
7	0	Erro no disco
	1	Nenhum erro no disco

O parâmetro *cdg_erro* contém as informações de erro que o DOS normalmente passaria para a rotina de tratamento de erro crítico no registrador DI. A Tabela 773.2 lista os valores que o DOS passa em DI para os erros de disco.

Tabela 773.2 Valores de erro no disco que o dos passa no registrador DI.

Valor	Significado	Valor	Significado
0	Protegido contra a gravação	7	Tipo de mídia desconhecido
1	Unidade desconhecida	8	Setor não-encontrado
2	Unidade não-preparada	9	Impressora sem papel
3	Comando desconhecido	10	Falha na gravação
4	Erro CRC nos dados	11	Falha na leitura
5	Estrutura de solicitação inválida	12	Falha geral
6	Erro de posic.onamento (*seek*)	15	Mudança de disco inválida

Finalmente, o parâmetro *cabecalho_dispo* é um ponteiro para o cabeçalho do controlador de dispositivo para o dispositivo que gerou o erro. Para lhe ajudar a compreender melhor como seus programas podem usar esses valores, o CD-ROM que acompanha este livro contém o seguinte programa, *errocrit.c*, que exibirá os valores que essas variáveis contêm:

```
int rotina(int errval,int ax,int bp,int si)
 {
   static char msg[80];
   unsigned di;
   int unidade, numerro;

   di= _DI;                  // Se isto não for um erro de
   if (ax < 0)
                             // disco então foi outro
                             // dispositivo tendo problema
    {
      erro_win("Erro no dispositivo"); // Informa o erro
      hardretn(ABORTAR);     // e retorna para o programa
                             // que diretamente pede Abortar
    }
   unidade = ax & 0x00FF;    // caso contrário, foi um erro
                             // no disco
   numerro = di & 0x00FF;    // Informa qual foi o erro
   sprintf(msg, "Erro: %s em %c\r\nA)bortar, R)epetir, I)gnorar: ",
       msg_erro[numerro], 'A' + unidade);
   hardresume(erro_win(msg)); // Retorna para o programa via
                              // interrupção 0x23 do DOS com
                              // Abortar, Repetir, ou Ignorar
                              // conforme a opção do usuário.
   return ABORTAR;
 }
```

774 RESTAURANDO AS INTERRUPÇÕES ALTERADAS

Quando seu programa termina, o DOS automaticamente restaura as definições da rotina de tratamento de interrupção Ctrl+Break, a rotina de tratamento de interrupçao do programa e as rotinas de tratamento de erro crítico com suas definições antes de você rodar o programa. Dependendo do seu programa, algumas vezes você desejará que o DOS restaure essas definições antes que seu programa termine Para restaurar as definições, muitos compiladores fornecem a função *_cexit*, como mostrado aqui:

```
#include <process.h>

void _cexit(void);
```

A função *_cexit* não termina seu programa. Em vez disso, ela simplesmente instrui o DOS a restaurar os vetores de interrupção que o parágrafo anterior discute. A função não fecha os arquivos nem descarrega os buffers de disco. Se seu compilador não fornecer a função *_cexit*, você poderá escrever uma função que restaure as interrupções usando as definições originais que seu programa salvou antes de alterar os vetores de interrupção.

775 CRIANDO UMA ROTINA DE TRATAMENTO DE CTRL+BREAK

Por padrão, quando o usuário pressionar a combinação de teclado Ctrl+Break, seu programa terminará. Normalmente, você não desejará que o usuário possa pressionar Ctrl+Break a qualquer momento para finalizar o programa. Como uma solução, seus programas podem definir uma rotina de tratamento de interrupção usando a função *ctrlbrk*, como mostrado aqui:

```
#include <dos.h>

void ctrlbrk(int (*rotina)(void));
```

Para criar sua própria rotina de tratamento de interrupção Ctrl+Break, defina uma função que você deseja que o programa chame toda vez que o usuário pressiona Ctrl+Break, e, depois, passa o nome da função para a função *ctrlbrk*. O programa a seguir, *ctrlbrk.c*, cria uma rotina de tratamento de Ctrl+Break personalizada:

```
#include <stdio.h>
#include <dos.h>

int Trata_Ctrl(void)
 {
   printf("\007Pressione Enter para finalizar o programa\n");
    return(1);
 }

void main(void)
 {
    ctrlbrk(Trata_Ctrl);
    printf("Pressione Enter para finalizar o programa\n");
    while (getchar() != '\n');
      ;
 }
```

O programa fica em um laço de repetição até que o usuário pressione Enter. Cada vez que o usuário pressionar Ctrl+Break, o programa chamará a função *Trata_Ctrl*. *Trata_Ctrl* soa o aviso sonoro e exibe uma mensagem que instrui o usuário a pressionar Enter para finalizar o programa. No programa *ctrlbrk.c*, a função retorna o valor 1. Se a rotina de tratamento retornar qualquer valor diferente de 0, o programa continuará. Se a rotina de tratamento retornar 0, o programa terminará.

776 USANDO OS SERVIÇOS DO DOS NA SUA ROTINA DE TRATAMENTO DE ERRO CRÍTICO

Quando o DOS chama uma rotina de tratamento de erro crítico, você precisa compreender que seu sistema está instável — um serviço do sistema operacional terminou abruptamente Dentro da sua rotina de tratamento de erro crítico, você deve restringir seu uso dos serviços do DOS para os serviços listados na Tabela 776.

Tabela 776 Serviços usáveis do DOS em uma rotina de tratamento de erro crítico

Serviço	Função	Serviço	Função
01H	Leitura de caractere	0BH	Teste do estado da entrada
02H	Escrita de caractere	0CH	Esvazia o buffer e recebe entrada
03H	Leitura pela porta auxiliar	3300H	Pega o status de Ctrl+C
04H	Escrita pela porta auxiliar	3301H	Define o status de Ctrl+C
05H	Escrita na impressora	3305H	Pega o disco de inicialização
06H	E/S direta no console	3306H	Lê a versão do DOS
07H	Leitura de caractere	50H	Define o PSP
08H	Leitura de caractere	51H	Obtém o PSP
09H	Escrita de string	59H	Lê o erro estendido
0AH	Leitura bufferizada do teclado	62H	Obtém o PSP

Se seus programas precisarem efetuar E/S dentro de uma rotina de tratamento de erro crítico, considere o uso de funções de E/S no arquivo de inclusão *conio.h*.

Aumentando o Desempenho com o Uso da Seleção do Conjunto de Instruções 777

Por padrão, a maioria dos compiladores baseados no DOS gera programas que podem rodar em todos os sistemas baseados nos processadores Intel, do 8088 até o Pentium. Caso saiba de antemão que os usuários rodarão um programa em uma máquina específica, você poderá aumentar o desempenho do programa usando o conjunto de instruções de uma máquina mais avançada. Por exemplo, o 80386 fornece instruções que não estão disponíveis no 8088. Usar uma dessas instruções do 80386 pode substituir várias instruções equivalentes do 8088. No entanto, quando você utilizar essas instruções, seus programas não rodarão mais nas máquinas antigas. Para gerar código executável para uma máquina específica, consulte as opções da linha de comando do seu compilador.

Funções Intrínsecas In-line 778

Para aumentar o desempenho, muitos compiladores C permitem que você substitua as funções por *código in-line*. Além de lhe permitir usar a palavra-chave *in-line* diante das funções que criar, muitos compiladores C lhe permitem substituir funções intrínsecas da biblioteca de execução por uma in-line correspondente. As funções intrínsecas que você pode inserir in line irão diferir de um compilador para outro. Consulte a documentação do compilador para determinar as funções disponíveis. No caso do compilador Borland C++, você pode colocar as funções intrínsecas listadas na Tabela 778 in-line.

Tabela 778 Funções intrínsecas que podem ser deixadas in-line que C suporta.

Funções intrínsecas que podem ser colocadas in-line						
alloc	*fabs*	*memchr*	*memcmp*	*memcpy*	*memset*	*rotl*
rotr	*stpcpy*	*strcat*	*strchr*	*strcmp*	*strcpy*	*strlen*
strncat	*strncmp*	*strncpy*	*strnset*	*strrchr*		

Para instruir seu compilador a colocar essas funções in-line, você pode usar uma opção da linha de comando ou a função *#pragma intrinsic*, discutida na Dica 779, a seguir.

Habilitando e Desabilitando as Funções Intrínsecas 779

Na dica anterior você aprendeu que muitos compiladores C lhe permitem substituir funções intrínsecas específicas com código in-line. Usando opções da linha de comando, você pode instruir o compilador a colocar as funções intrínsecas in-line. Além disso, muitos pré-processadores suportam a diretiva *pragma intrinsic*, o que lhe permite habilitar ou desabilitar as funções intrínsecas in-line, como mostrado aqui:

```
#pragma  intrinsic função      // Habilita in-line
#pragma  intrinsic -função     // Desabilita in-line
```

Por exemplo, o comando a seguir instrui o compilador a gerar código in-line para a função *strlen:*

```
#pragma  intrinsic strlen
```

Quando você usa pragma intrinsic, precisa preceder pragma com um protótipo de função. Quando o compilador encontrar *pragma*, ele substituirá o nome da função com um nome equivalente que inicia e termina com um traço de sublinhado. No caso da função *strlen*, o compilador gerará a constante *_strlen_*, como mostrado aqui:

```
#define strlen _strlen_
```

780 COMPREENDENDO AS CHAMADAS RÁPIDAS DE FUNÇÕES

Quando seu programa chama uma função, C passa parâmetros para a função usando a pilha. Como discutido na seção Funções, anteriormente, o uso da pilha consome muito tempo da função. Para tentar tornar essas chamadas mais rápidas, alguns compiladores C fornecem um modificador *_fastcall* que você pode colocar diante de um nome de função, como mostrado aqui:

```
int _fastcall umafuncao(int a, int b);
```

O programa a seguir, *fastcall.c*, ilustra o modificador *_fastcall:*

```
#include <stdio.h>
#include <time.h>

int _fastcall soma_rapida(int a, int b)
 {
   return(a + b);
 }

int soma_lenta(int a, int b)
 {
   return(a + b);
 }

void main(void)
 {
   unsigned long int i, result;

   clock_t hora_inicial, hora_final;
   printf("Processando...\n");
   hora_inicial = clock();
   for (i = 0; i < 2000000L; i++)
     result = soma_rapida(i, -i);
   hora_final = clock();
   printf("Tempo de processamento para chamada rápida %d ticks\n", hora_
   final - hora_inicial);
   hora_inicial = clock();
   for (i = 0; i < 2000000L; i++)
     result = soma_lenta(i, -i);
   hora_final = clock();
   printf("Tempo de processamento para a chamada normal %d ticks\n",
   hora_final - hora_inicial);
 }
```

781 REGRAS PARA A PASSAGEM DE PARÂMETROS _FASTCALL

Na dica anterior você aprendeu que muitos compiladores suportam o modificador de função *_fastcall*, o que instrui o compilador a passar parâmetros para a função usando registradores. Dependendo da máquina-alvo, o número de registradores disponíveis para os parâmetros diferirá. No caso do compilador Borland C++, seus programas somente podem passar três parâmetros via registradores. A Tabela 781 especifica como o modificador *_fastcall*, quando usado com o compilador Borland C++, passa parâmetros para as funções.

Tabela 781 Registradores usados para a passagem de parâmetros com o modificador *_fastcall*.

Tipo de Parâmetro	Registradores Usados
char (sinalizado e não-sinalizado)	AL, DL, BL
int (sinalizado e não-sinalizado)	AX, DX, BX
long (sinalizado e não-sinalizado)	DX, AX
ponteiro *near*	AX, DX, BX
outros	Passados na pilha

COMPREENDENDO O CÓDIGO INVARIANTE 782

À medida que você examina as diretivas do compilador que afetam a otimização, pode encontrar o termo *código invariante*. Em geral, o código invariante refere-se aos comandos que aparecem dentro de um laço cujos valores não mudam. Por exemplo, o laço *for* a seguir atribui o resultado da multiplicação *a* * *b* * *c* a cada elemento da matriz.

```
for (i = 0; i < 100; i++)
    matriz[i] = a * b * c;
```

Como as variáveis *a*, *b* e *c* não mudam dentro do laço, o resultado da multiplicação é *invariante* (não muda). À medida que você programa, deve ficar atento ao código invariante. Ao encontrar código invariante, você pode normalmente aumentar o desempenho modificando de alguma forma o programa. No caso do laço *for* anterior, você pode aumentar o desempenho do programa substituindo a multiplicação pelo seu resultado, como mostrado aqui:

```
resultado = a * b * c;
for (i = 0; i < 100; i++)
    matriz[i] = resultado;
```

Para melhorar o desempenho do programa, muitos compiladores verificam a existência de código invariante, substituindo-o dentro de código-objeto-alvo com um equivalente que não seja invariante. Idealmente, você mesmo deve localizar e corrigir o código invariante. No entanto, usando as opções da linha de comando, você poderá instruir seu compilador a efetuar as substituições para você durante a compilação.

COMPREENDENDO A SUPRESSÃO DE CARGA REDUNDANTE 783

Como você aprendeu, para melhorar o desempenho, o compilador C freqüentemente carrega valores nos registradores. Quando o compilador efetua a *supressão de carga redundante*, ele controla os valores que já colocou nos registradores. O compilador então referencia os registradores quando precisa de um valor, em vez de carregar o valor novamente. O compilador usa a supressaão da carga redundante para evitar as operações de carga duplicadas, o que melhora o desempenho do programa. A desvantagem em usar a supressão de carga é que a compilação torna-se um pouco mais demorada. Entretanto, como regra, você deve sempre instruir o compilador a efetuar a supressão da carga.

COMPREENDENDO A COMPACTAÇÃO DO CÓDIGO 784

Quando você examinar a documentação do seu compilador, poderá encontrar o termo *compactação do código*. Em geral, a compactação do código usa desvios para o código anterior para eliminar os comandos redundantes. Por exemplo, considere o seguinte programa, *compacto.c*:

```
#include <stdio.h>

void main(void)
 {
   int a = 1, b, c, d;

   switch (a) {
    case 1: a = 5;
            b = 6;
            c = 7;
            d = 8;
```

```
            break;
    case 2: b = 6;
            c = 7;
            d = 8;
            break;
    };
}
```

Se você examinar o comando *switch*, verá que os comandos que o programa executa para cada *case* são muito similares. Em vez de duplicar os comandos de atribuição em ambas as posições, o compilador poderia colocar uma instrução JMP (que equivale na linguagem assembly ao comando *goto* do basic) que salta de volta ao comando *b = 6*, que ocorre no primeiro *case* e no início do segundo *case*.

785 COMPREENDENDO A COMPACTAÇÃO DO LAÇO

Se você examinar os laços *for* que ocorrem em seus programas, verificará que muitos laços manipulam uma string ou outra matriz. Quando seu programa atribui o mesmo valor a todo elemento em uma matriz, o compilador C pode otimizar o desempenho do seu programa substituindo o laço com uma das instruções ST*xxx* do 80x86. Por exemplo, o laço *for* a seguir inicializa a matriz *string_nula* com NULL:

```
for (i = 0; i < sizeof(string_nula); i++)
    string_nula[i] = NULL;
```

Se você examinar a listagem em linguagem assembly gerada pelo compilador, verá que ele elimina o laço, usando a instrução *STOSW*. Esse tipo de substituição que o compilador efetua é chamado *compactação do laço*.

786 COMPREENDENDO A INDUÇÃO DO LAÇO E A REDUÇÃO DA FORÇA

A indução do laço e a redução da força são técnicas que o compilador usa para otimizar os laços dentro de um programa. O compilador normalmente otimiza os laços dentro de um programa quando o programa manipula matrizes dentro de um laço. Por exemplo, considere o laço a seguir, que atribui valores aos elementos da matriz:

```
for (i = 0; i < 128; i++)
    matriz[i] = 0;
```

Para cada referência à matriz, o compilador precisa efetuar uma operação de multiplicação para determinar o elemento correto (*base + i * sizeof(tipo_matriz)*). Em vez de usar a matriz, o compilador pode usar um ponteiro, como mostrado aqui:

```
ender = &matriz[128];

for (ptr = matriz; ptr < ender; ptr++)
    *ptr = 0;
```

Eliminando a lenta operação de multiplicação, o compilador melhora o desempenho do programa. O processo de criar novas variáveis a partir de variáveis do laço é chamado *indução do laço*. Como as variáveis induzidas são geralmente menos complexas que as variáveis que elas substituem, o laço recém-criado introduz uma *redução de força*.

787 COMPREENDENDO A ELIMINAÇÃO DE SUBEXPRESSÃO COMUM

Caso seus programas trabalhem com matrizes, algumas vezes você poderá ignorar o desempenho do seu programa eliminando as subexpressões comuns. Por exemplo, considere o seguinte comando *if*, que testa se um elemento da matriz contém uma letra A maiúscula ou minúscula:

```
if ((matriz[i] == 'A') || (matriz[i] == 'a'))
```

Para cada teste, o compilador precisa determinar o elemento da matriz executando uma multiplicação *(base + i * sizeof(tipo_matriz))*. No entanto, uma implementação mais rápida substituiria a referência da matriz por um ponteiro, como mostrado aqui:

```
ptr = &matriz[i];
if ((*ptr == 'A') || (*ptr == 'a'))
```

Substituir a subexpressão comum com o código alternativo mais rápido melhora o desempenho do programa. No entanto, em alguns casos, tentar eliminar as subexpressões comuns pode tornar seus programas mais difíceis de compreender. Muitos compiladores C irão detectar as oportunidades para a eliminação da subexpressão e irão efetuar as operações para você dentro do código resultante (compilado). À medida que você trabalhar com condições compostas, tenha em mente que poderá melhorar o desempenho do programa eliminando ou reduzindo as subexpressões.

COMPREENDENDO AS CONVERSÕES COMUNS DE C PADRÃO 788

Quando você efetua operações aritméticas em diferentes tipos de valores, o compilador C geralmente promove o valor de tipo inferior. Para lhe ajudar a compreender as conversões padrão de C, considere as regras a seguir, que C aplica na ordem do primeiro para o último item nessa lista:

- Com a exceção de *unsigned short*, o compilador C promove todos os valores inteiros pequenos para *int*. O compilador C promove os valores do tipo *unsigned short* para *unsigned int*.
- Se um dos operandos *for long double*, o compilador C promove o outro para *long double*.
- Se um dos operandos for *double*, o compilador C promove o outro para *double*.
- Se um dos operandos for *float*, o compilador C promove o outro para *float*.
- Se um dos operandos for *unsigned long*, o compilador C promove o outro para *unsigned long*.
- Se um dos operandos for *long*, o compilador promove o outro para *long*.
- Se um dos operandos for *unsigned*, o compilador C promove o outro para *unsigned*.
- Caso contrário, o compilador C trata ambos os operandos como tipo *int*.

COMPREENDENDO OS QUATRO TIPOS BÁSICOS DE C 789

À medida que você examina as declarações complexas em C, tenha em mente que C suporta quatro tipos básicos: *void, scalar, function* e *aggregate*. O tipo *void* especifica a ausência de valores. Por exemplo, *void*, em uma lista de parâmetros, informa que uma função não recebe parâmetros. Da mesma forma, *void*, diante de um nome de função, especifica que a função não retorna um valor. Os valores *scalar* incluem aritmético, enumerado, ponteiro e valores de referência. Um tipo *function* especifica uma função que retorna um tipo específico. Finalmente, um tipo *aggregate* especifica uma matriz, união, estrutura ou classe de C++. À medida que você for examinando declarações complexas, tente mapear a declaração para um dos quatro tipos que C suporta.

COMPREENDENDO OS TIPOS FUNDAMENTAIS VERSUS OS DERIVADOS 790

Ao examinar declarações complexas em C, compreenda que C suporta os tipos *fundamentais* e *os derivados*. Os tipos de dados fundamentais de C incluem os seguintes: *void, char, double, float* e *int*. Além disso, C lhe permite aplicar os modificadores *long, short, signed* e *unsigned* aos tipos fundamentais. Os tipos derivados, por outro lado, incluem matrizes, classes, funções, ponteiros, estruturas e uniões para os outros tipos. Os tipos *char, int, long* e *short* são tipos integrais. À medida que você for examinando as declarações, os seguintes tipos integrais serão equivalentes:

```
char, signed char     // Normalmente um compilador padrão
int, signed int
unsigned, unsigned int
short, short int, signed short int
unsigned short, unsigned short int
long, long int, signed long int
unsigned long, unsigned long int
```

791 COMPREENDENDO OS INICIALIZADORES

Inicializadores são os valores que seus programas atribuem para as variáveis na declaração. Quando você usar inicializadores, lembre-se das seguintes regras:

- Se um programa não inicializa explicitamente um tipo aritmético, a maioria dos compiladores inicializará a variável com 0.
- Se um programa não inicializa explicitamente um tipo de ponteiro, a maioria dos compiladores inicializará o ponteiro com *NULL*.
- Se o número de inicializadores exceder o número de variáveis a ser inicializado, o compilador gerará um erro.
- Todas as expressões que você usa para inicializar uma variável precisam ser constantes (C++ não requer isso) se os inicializadores forem para um objeto estático, matriz, estrutura ou união
- Se a variável declarada tem escopo no bloco, e o programa não declarou a variável como externa, então a declaração não pode ter um inicializador.
- Se o código do programa fornecer menos inicializadores que o compilador requer para inicializar totalmente a variável, o compilador inicializará o restante dos valores seguindo a técnica de inicialização padrão.

792 COMPREENDENDO O SISTEMA DE LIGAÇÃO (LINKAGE)

Como você aprendeu, o linkeditor combina código de seus programas, arquivos-objetos e bibliotecas. Dependendo do linkeditor que você usa dentro de uma dada compilação, algumas vezes, duas ou mais funções nos arquivos ligados têm o mesmo nome. *Linkage* é o processo que determina qual função o linkeditor aplicou a uma referência. Em C, os identificadores têm um dentre três atributos de ligação possíveis: *external, internal* e *none*. Um identificador com linkage *external* representa o mesmo objeto em todos os arquivos que compõem um programa. Um identificador com *linkage internal* representa o mesmo objeto dentro de um arquivo. Um identificador com *no linkage* é exclusivo dentro dos arquivos — significando que ele ocorre somente uma vez. O linkeditor determina com qual função C associa o identificador. O linkeditor usa as seguintes regras de *linkage:*

- Os identificadores declarados como *static* têm *linkage internal*.
- Se um identificador aparecer com ligação interna ou externa, C usará *linkage internal*, e C++ usará *linkage external*.
- Se o programa declarar um identificador com a palavra-chave *extern*, o identificador terá a mesma ligação que qualquer declaração visível com escopo em arquivo; se essa declaração não existir, o identificador terá *linkage external*.
- Se o identificador de uma função não tiver especificador de classe de armazenamento, o identificador terá a mesma ligação como se o programa usasse *extern* com o identificador.
- Os identificadores de objeto declarados sem um especificador de classe de armazenagem têm *linkage external*.
- Os identificadores que você declara para ser algo diferente de um objeto ou de uma função não têm *linkage*.
- Os parâmetros de função não têm *linkage*.
- Os identificadores que têm escopo de bloco declarado sem uma classe de armazenagem *extern* não têm *linkage*.

793 COMPREENDENDO AS DECLARAÇÕES EXPERIMENTAIS

Uma *declaração experimental* é uma declaração de dados externos que não tem especificador de classe de armazenamento e nenhuma inicialização. Por exemplo, assuma que o compilador encontre a seguinte declaração:

```
int valor;
```

Se o compilador, mais tarde, encontrar uma definição para a variável, ele tratará a variável como se a palavra-chave *extern* a precedesse. Se o compilador atingir o final da unidade de tradução sem encontrar uma definição, ele alocará memória para a variável. O programa a seguir, *experim.c*, cria uma declaração de variável experimental para a variável *valor:*

```
#include <stdio.h>

int valor;

void main(void)
 {
   printf("%d\n", valor);
 }

int valor = 1500;
```

Quando o compilador encontrar a definição da variável *valor* que inicializa a variável como 1500, ele converterá a primeira declaração de *valor* de uma definição experimental para uma definição completa.

Contrastando Declarações e Definições 794

Muitas dicas apresentadas neste livro usam os termos "declaração" e "definição". Em geral, uma *declaração* introduz um ou mais identificadores dentro de um programa. Uma definição, por outro lado, instrui o compilador a realmente alocar memória para o objeto. Por exemplo, você pode considerar um protótipo de função como uma declaração, e o cabeçalho da função e o código como uma definição. C classifica as declarações como *defining* (de definição) ou *referencing* (de referência). Uma declaração de *definição* declara um ou mais identificadores, e define a quantidade de memória que o compilador deve alocar para o objeto. Uma declaração de *referência*, por outro lado, simplesmente introduz um identificador. Dentro de C, você pode declarar os objetos listados na Tabela 794.

Tabela 794 Tipos declaráveis em C.

Objetos			
matrizes	classes	membros de classe	enumerados
tags enumerados	constantes	funções	rótulos
macros	estruturas	membros de estruturas	tipos
uniões	membros da união	variáveis	

Compreendendo lvalues 795

Quando seu programa trabalha com ponteiros, você pode encontrar mensagens de erro do compilador que dizem que o compilador requer um *lvalue*. Um *lvalue* é uma expressão que o compilador pode usar para localizar um objeto. Você pode considerar *lvalues* como expressões que seriam válidas no lado esquerdo do operador de atribuição. Os seguintes são *lvalues* válidos:

```
variavel = valor;

*variavel = valor;

variavel[1] = valor;
```

No entanto, é importante observar que cada um dos *lvalues* que acabam de ser mostrados poderiam também estar no lado direito do operador de atribuição. Um *lvalue* simplesmente fornece um valor que o compilador pode usar para localizar um objeto na memória. C suporta *lvalues* modificáveis e não-modificáveis. Um *lvalue* modificável é um valor de ponteiro que você pode alterar para apontar para um valor diferente. A maioria dos compiladores que você usou neste livro são *lvalues* modificáveis. Por outro lado, você não pode modificar um *lvalue* não-modificável. Por exemplo, uma constante ponteiro é um *lvalue* não-modificável.

Compreendendo rvalues 796

Quando você compila seus programas, pode encontrar uma mensagem de erro que diz que o compilador encontrou um *rvalue* inesperado. Um *rvalue* é uma expressão que aparece no lado direito de um sinal de igual. As expressões a seguir são exemplos de *rvalues:*

```
resultado = valor;

resultado = 1001;

resultado = valor + 1500;
```

No entanto, os comandos a seguir são inválidos porque não especificam uma posição de memória na qual C pode atribuir um valor:

```
1500 = resultado;
valor + 1500 = resultado;
```

Esses erros de declaração normalmente aparecem quando os usuários estão tentando criar ponteiros, como mostrado aqui:

```
*(valor + 1500) = resultado;
```

797 USANDO PALAVRAS-CHAVE DE REGISTRADORES DE SEGMENTO

Como você aprendeu, o PC usa quatro registradores específicos para localizar o código, os dados e a pilha do seu programa. Seus programas podem usar a função da biblioteca de execução *segread* para determinar as definições do registrador. Além disso, muitos compiladores baseados no DOS fornecem as palavras-chave listadas na Tabela 797.

Tabela 797 Palavras-chave adicionais para registradores que muitos compiladores baseados no DOS oferecem.

Palavra-chave	Significado
_cs	Cria um ponteiro para o segmento de código
_ds	Cria um ponteiro para o segmento de dados
_es	Cria um ponteiro para o segmento extra
_ss	Cria um ponteiro para o segmento de pilha

A declaração a seguir cria um ponteiro para o segmento de pilha:

```
char _ss *meu_ponteiro_pilha;
```

Dependendo do modelo de memória atual, os ponteiros irão conter ponteiros near ou far, conforme for apropriado.

798 USE OS PONTEIROS FAR COM CUIDADO

Na seção Matrizes, Ponteiros e Estruturas, anteriormente, você aprendeu que um ponteiro far é um ponteiro de 32 bits que contém um segmento de 16 bits e um endereço de deslocamento de 16 bits. Os ponteiros far permitem que seus programas acessem o intervalo de 1Mb da memória convencional. No entanto, quando você usa ponteiros far, precisa compreender como o valor do ponteiro volta a zero quando o valor de deslocamento excede seu limite de 16 bits. Por exemplo, assuma que o ponteiro far *char local* contenha o seguinte valor:

```
local = 0x1000FFFE;   // Segmento 0x1000 Deslocamento FFFE
```

Se você incrementar o ponteiro, o valor dele se tornará o seguinte:

```
local = 0x1000FFFF;   // Segmento 0x1000 Deslocamento FFFF
```

Se você incrementar o ponteiro de novo, um erro possivelmente ocorrerá, como mostrado aqui:

```
local = 0x10000000;   // Segmento 0x1000 Deslocamento 0000
```

Observe que o valor do deslocamento voltou para 0, mas o valor do segmento não mudou. Como resultado do processo de incrementar, o ponteiro voltou para o início do endereço do segmento de 64Kb. Se você precisar que o ponteiro se mova para o início do próximo segmento, use um ponteiro *huge*.

COMPREENDENDO OS PONTEIROS NORMALIZADOS 799

Como você sabe, o PC endereça as posições de memória usando endereços de segmento e de deslocamento. O PC permite até 65.536 endereços de segmento. Cada endereço de segmento inicia em um endereço de 16 bytes: 0, 16, 32, 48 e assim por diante. Se você multiplicar o endereço de 16 bytes pelos 65.536 segmentos, o resultado é um espaço de endereço de 1Mb. Dado um endereço de segmento, um endereço de deslocamento lhe permite escolher uma das 65.536 posições possíveis dentro do segmento. Quando você usa endereços de segmento e de deslocamento, pode endereçar cada posição na memória usando diferentes combinações de endereço e de segmento. Por exemplo, assuma que você queira endereçar a posição 48 na memória. Para fazer isso, você pode usar qualquer uma das seguintes combinações de segmento/endereço:

- Segmento 0 Deslocamento 48
- Segmento 1 Deslocamento 32
- Segmento 2 Deslocamento 16
- Segmento 3 Deslocamento 0

Como você pode referenciar cada posição de memória diferentemente, é possível que dois ponteiros far referenciem a mesma posição de memória, mas contenham diferentes valores. Considere as seguintes atribuições de ponteiro:

```
char far *ptr1 = 0x00000030;   // Seg 0 deslocamento 48
char far *ptr2 = 0x00030000;   // Seg 3 deslocamento 0
```

Ambos os ponteiros irão referenciar a mesma posição de memória. No entanto, caso seu programa compare os valores dos ponteiros, os valores não são iguais. Um ponteiro *normalizado* elimina essa disparidade armazenando sempre os valores de modo que o compilador use um deslocamento de 16 bytes. Assim, o compilador sempre armazena endereços de segmento usando o segmento mais próximo do valor. No caso anterior, o ponteiro normalizado conteria o segmento 3 e o deslocamento 0

COMANDOS DO CO-PROCESSADOR MATEMÁTICO 800

Um processador de ponto flutuante (ou matemático) é um circuito integrado especializado que contém instruções que podem efetuar operações aritméticas rapidamente — tais como divisão, multiplicação e até cálculo da raiz quadrada —, usando valores em ponto flutuante. Caso esteja usando um 8088, 80286 ou 80386, você precisará adquirir um co-processador de ponto flutuante (um 8087, 80287 ou 80387). Se estiver usando um computador equipado com o processador 80486DX ou posterior, o co-processador matemático estará embutido no processador principal. Como os co-processadores matemáticos somente efetuam operações de ponto flutuante, eles podem efetuar as operações rapidamente. Caso seu computador tenha um co-processador matemático, você deverá instruir o compilador a gerar instruções que usem o co-processador. Dependendo do seu compilador, as *opções que você precisará usar para gerar instruções para um co-processador de ponto flutuante diferirão.* Para permitir que os programas rodem em sistemas que não têm um co-processador matemático, normalmente a maioria dos compiladores não gera instruções em ponto flutuante.

Se você estiver usando o *Turbo C++ Lite*, o compilador suportará várias opções */FPx* que lhe permitem instruir o compilador sempre a usar as instruções em ponto flutuante. As opções de ponto flutuante que você seleciona afetarão o tamanho e a velocidade do programa. Consulte a documentação do compilador para obter maiores informações sobre as opções de ponto flutuante.

COMPREENDENDO CDECL E PASCAL NAS VARIÁVEIS 801

À medida que você examinar programas que usam módulos mistos de linguagens, tais como Pascal e C, poderá encontrar variáveis declaradas com os modificadores *pascal* e *cdecl*. Para manter a compatibilidade com os identificadores Pascal, o modificador *pascal* instrui o compilador a ignorar a não-distinção da caixa das letras e a não preceder o identificador com um traço de sublinhado. O modificador *cdecl*, por outro lado, instrui o compilador a distinguir a caixa das letras e a incluir os traços sublinhados preliminares. Por exemplo, o programa a seguir declara uma variável externa chamada *numero* que um programa Pascal define:

```
#include <stdio.h>
```

```
extern int pascal numero;
void main(void)
 {
   printf("O valor é %d", numero);
 }
```

802 IMPEDINDO AS INCLUSÕES CIRCULARES

À medida que seus programas fizerem amplo uso dos arquivos de cabeçalho (os programas C++ normalmente definem as classes em arquivos de cabeçalho), algumas vezes um arquivo que você incluir, incluirá um segundo arquivo de cabeçalho, o qual, por sua vez, incluirá o primeiro arquivo de cabeçalho. Quando o pré-processador efetuar as inclusões, ele terminará em uma operação circular. Para reduzir a possibilidade de operações circulares, seus arquivos de cabeçalho podem declarar uma macro que impeça o compilador de processar os arquivos uma segunda vez. Por exemplo, o arquivo de cabeçalho a seguir usa a macro *MINHAS_COISAS_DEF* para determinar se o compilador já processou o conteúdo:

```
#ifndef MINHAS_COISAS_DEF
#define MINHAS_COISAS_DEF 1

 //   Outros comandos de inclusão

#endif
```

Neste caso, a primeira vez que o compilador processar o arquivo de cabeçalho, ele definirá uma macro. Se o programa incluir o arquivo de cabeçalho uma segunda vez, ele não processará o conteúdo do arquivo, devido à diretiva *#ifndef*.

803 INTRODUZINDO C++

C++ é uma linguagem de programação desenvolvida pelo Dr. Bjarne Stroustrup no Bell Labs, da AT&T, que expande a linguagem C acrescentando capacidades de orientação a objetos e outras melhorias. Você pode pensar em C++ como um superconjunto de C, pois C++ suporta os recursos da linguagem C que você já aprendeu neste livro. No entanto, como você aprenderá, C++ é mais do que "um C orientado a objetos" — C++, na verdade acrescenta muitos novos recursos que aumentarão as capacidades do seu programa. Se você estiver usando um compilador C++ (tal como *Turbo C++ Lite*), a maioria dos programas apresentados nas 802 dicas anteriores deverão compilar e executar sem quaisquer modificações. As dicas restantes iniciam com os fundamentos de C++, e expandem o conhecimento que você adquiriu em C. Quando você chegar ao final deste livro, deverá dominar C e C++.

804 COMO OS ARQUIVOS-FONTE DE C++ SÃO DIFERENTES

Em geral, não há diferenças entre os arquivos-fonte C e C++. Ambas as linguagens suportam plenamente as diretivas do compilador, tais como *#include* e *#define*. Com relação à nomeação, muitos programadores usam a extensão CPP para diferenciar os arquivos-fonte C e C++. Tudo o que você aprendeu até aqui ainda será aplicado quando você começar a criar programas C++, com a exceção de algumas estruturas e constantes exclusivas. Isto é, em C++, você ainda pode incluir arquivos de cabeçalho, vincular bibliotecas de código-objeto, e assim por diante.

805 INICIANDO COM UM PROGRAMA SIMPLES EM C++

Na Dica 2, você criou seu primeiro programa C, o qual usava *printf* para exibir uma mensagem na tela, como mostrado aqui:

```
#include <stdio.h>

void main(void)
 {
   printf("Bíblia do Programador C/C++, do Jamsa!");
 }
```

O programa C++ a seguir, *simples.cpp*, realiza um processamento idêntico:

```
#include <iostream.h>
void main(void)
 {
   cout << "Bíblia do Programador C/C++, do Jamsa!";
 }
```

O programa *simples.cpp* usa o canal de E/S *cout* de C++, discutido na dica a seguir. Você pode compilar e construir o programa *simples.c* exatamente como criou todos os seus programas C anteriormente. Quando você executar o programa *simples.cpp*, sua tela exibirá o seguinte:

```
Bíblia do Programador C/C++, do Jamsa!
C:\>
```

COMPREENDENDO O CANAL DE ENTRADA/SAÍDA COUT 806

Na dica anterior, o programa de exemplo usou o canal de E/S *cout* para escrever uma string de caracteres na tela, como mostrado aqui:

```
cout << "Bíblia do Programador C/C++, do Jamsa!"
```

Redirecionar a saída para o canal de E/S *cout* é o mesmo que usar *printf* para escrever saída em *stdout*. O símbolo << não é o *operador de deslocamento à esquerda bit a bit*. Em vez disso, o símbolo é o operador de saída que especifica para qual canal o programa envia os dados. O programa a seguir, *cout.cpp*, usa o operador de saída de C++ para exibir várias mensagens diferentes:

```
#include <iostream.h>

void main(void)
 {
   cout << "Esta é a linha 1.\n";
   cout << "Este texto está na ";
   cout << "linha 2.\n";
   cout << "Esta é a última linha.";
}
```

Quando você compilar e executar o programa *cout.cpp*, sua tela exibirá o seguinte:

```
Esta é a linha 1.
Este texto está na linha 2.
Esta é a última linha.
C:\>
```

ESCREVENDO VALORES E VARIÁVEIS COM COUT 807

Como você aprendeu, o canal de saída *cout* permite que seu programa exiba saída na tela de texto. As dicas anteriores usaram *cout* paa exibir as strings de caracteres. O programa a seguir, *cout_num.cpp*, usa *cout* para exibir strings de caracteres e números:

```
#include <iostream.h>

void main(void)
 {
   cout << "cout permite a exibição de strings, ints e *floats\n";

   cout << 1500;
   cout << "\n";
   cout << 1.2345;
 }
```

Quando você compilar e executar o programa *cout_num.cpp*, sua tela exibirá a seguinte saída:

```
cout permite a exibição de strings, ints, e floats
1500
1.2345
C:\>
```

808 COMBINANDO DIFERENTES TIPOS DE VALORES COM COUT

Você viu na dica anterior que o canal de E/S *cout* permite que seus programas exibam todos os tipos de valores. O programa *cout_num.cpp*, apresentado na Dica 807, usou diversos comandos para exibir sua saída, como mostrado aqui:

```
cout << "cout permite a exibição de strings, ints, e floats\n";
cout << 1500;
cout << "\n"; cout <<
     <<1.2345;
```

Felizmente, *cout* lhe permite colocar tipos de valores diferentes no canal de saída em um comando, como mostrado no seguinte programa, *cout_um.cpp*.

```
#include <iostream.h>

void main(void)
 {
      cout << "cout exibe strings " << 1500 << "\n" << 1.2345;
 }
```

809 EXIBINDO VALORES HEXADECIMAIS E OCTAIS

Como você aprendeu, o canal de E/S *cout* permite que seus programas exibam valores do tipo *int* e *float*. Ao usar *printf* para exibir valores inteiros, você pode usar os especificadores de formato *%x* e *%o* para exibir os valores em hexadecimal e octal. Quando seus programas usam *cout* para exibir saída, eles podem usar os modificadores *dec*, *oct* e *hex*, como mostrado no programa a seguir, *cout_hex.c*:

```
#include <iostream.h>

void main(void)
 {
   cout << "Valor decimal " << dec << 0xFF;
   cout << "\nValor octal " << oct << 10;
   cout << "\nvalor hexadecimal " << hex << 255;
 }
```

810 REDIRECIONANDO COUT

Seus programas podem usar o canal de E/S *cout* para exibir a saída, exatamente como se tivessem escrito a saída diretamente em *stdout*. Portanto, se você tem um programa que usa *cout*, pode redirecionar a saída do programa para um arquivo ou para algum outro dispositivo. O programa a seguir, *1_a_100.cpp*, usa *cout* para exibir os números de 1 a 100:

```
#include <iostream.h>

void main(void)
 {
   int i;

   for (i = 1; i <= 100; i++)
     cout << i  << '\n';
 }
```

Usando o operador de redirecionamento da saída do DOS, você pode redirecionar a saída do programa para um arquivo, como mostrado aqui:

```
C:\> 1_A_100  >> NOMEARQ.EXT     <Enter>
```

SE VOCÊ GOSTA DE PRINTF, USE PRINTF 811

Várias dicas anteriores realizaram sua saída usando o canal de E/S *cout*. Se você se sentir mais à vontade usando a função *printf*, use *printf*. Em dicas posteriores, você aprenderá como formatar a saída que seu programa exibe com *cout*. A partir desse momento você preferirá usar *cout* para todas as suas saídas. No entanto, como regra, para tornar seus programas mais fáceis de compreender, você deverá escolher uma técnica e usar somente ela. O programa a seguir, *ambos.cpp*, exibe saída usando *cout* e *printf*:

```
#include <iostream.h>
#include <stdio.h>

void main(void)
 {
   cout << "Bíblia do ";
   printf("programador C/C++, ");
   cout << "do Jamsa!";
 }
```

ESCREVENDO SAÍDA EM CERR 812

Como você sabe, quando seus programas escrevem a saída no indicativo de arquivo *stderr*, C++ não pode redirecionar a saída da tela. Se você estiver usando os canais de E/S de C++ para realizar entrada e saída, seus programas poderão escrever a saída no canal de E/S *cerr*. O programa a seguir, *usa_cerr.cpp*, usa *cerr* para impedir que o usuário ou o programa redirecione a saída do programa.

```
#include <iostream.h>

void main(void)
 {
   int i;

   for (i = 1; i <= 100; i++)
     cerr << "Não é possível redirecionar cerr " << i  << '\n';
 }
```

RECEBENDO ENTRADA COM CIN 813

O canal de E/S *cout* permite que seus programas exibam saída em *stdout*. De um modo similar, seus programas C++ podem receber entrada usando o canal de E/S *cin*. O programa a seguir, *usa_cin.cpp*, usa *cin* para receber entrada a partir de vários tipos diferentes de variáveis:

```
#include <iostream.h>

void main(void)
 {
   int idade;
   float salario;
   char nome[128];

   cout << "Digite seu nome idade salário: ";
   cin >> nome >> idade >> salario;
   cout << nome << ' ' << idade << ' ' << salario;
 }
```

CIN NÃO USA PONTEIROS 814

Como você viu na dica anterior, o canal *cin* não usa ponteiros para referenciar as variáveis. Ao receber informações com uma função tal como *scanf*, você precisará passar explicitamente um ponteiro para a variável que receberá as informações (para que a função possa modificar a variável). No entanto, por causa da construção do

canal *cin*, você não deverá passar um ponteiro para uma variável para *cin* — se fizer isso, *cin* retornará um erro. Por exemplo, o programa a seguir, *cin_err.cpp* retornará seis erros quando você tentar compilar o programa:

```
#include <iostream.h>

void main(void)
 {
   int idade;
   float salario;
   char nome[128];

   cout << "Digite seu nome idade salario: ";
   cin >> &nome >> &idade >> &salario;
   cout << nome << ' ' << idade << ' ' << salario;
 }
```

Você aprenderá em dicas posteriores como *cin* coloca valores dentro de uma variável não-ponteiro. Por enquanto, simplesmente compreenda que seus programas precisam passar nomes de variáveis reais para *cin*.

815 COMPREENDENDO COMO CIN SELECIONA OS CAMPOS DE DADOS

Na Dica 813, você usou o canal de E/S *cin* para ler o nome, idade e salário do usuário em uma linha, como mostrado aqui:

```
cin >> nome >> idade >> salario;
```

Quando seus programas usam *cin* para ler entrada, você precisa compreender como *cin* analisa a entrada. A não ser que informado de outra forma, *cin* usa espaço em branco (um branco, tabulação ou nova linha) para delimitar os campos de entrada. Portanto, se o usuário digitasse seu nome completo (tal como João Silva) na operação anterior, *cin* usaria o primeiro nome para a variável *nome* e o último nome para a variável *idade*, e a operação de E/S estaria com erro. Em dicas posteriores você aprenderá como efetuar entrada formatada usando *cin*.

816 COMPREENDENDO COMO OS CANAIS DE E/S CONHECEM OS TIPOS DOS VALORES

Como você aprendeu, seus programas podem usar os canais de E/S *cin*, *cout* e *cerr* para efetuar operações de E/S para os indicativos de arquivo *stdin*, *stdout* e *stderr*. Usando esses canais de E/S, você pode efetuar as operações de E/S com strings de caracteres, valores inteiros e de ponto flutuante. Quando seus programas efetuam as operações de E/S usando *printf* e *scanf*, as funções usam os especificadores de formato para determinar os tipos de valores (tais como *string*, *int* e assim por diante). Quando você executa as operações de entrada e saída com os canais de E/S de C++, o compilador fornece informações sobre o tipo de cada valor, de modo que não há necessidade de um especificador de formato. Como um exercício interessante, você pode gerar e examinar uma listagem em linguagem assembly de um arquivo, tal como *use_cin.cpp*, que usa *cin* e *cout*, caso seu compilador gere essa listagem.

817 EFETUANDO SAÍDA USANDO CLOG

Você já sabe que, C++ fornece os canais de E/S *cin*, *cout* e *cerr* que correspondem a *stdin*, *stdout* e *stderr*. Além disso, C++ fornece um quarto canal de E/S chamado *clog*. O canal de E/S *clog* é similar a *cerr*, exceto que efetua saída bufferizada. O programa a seguir, *clog.cpp*, usa o canal de E/S *clog* para exibir uma mensagem:

```
#include <iostream.h>

void main(void)
 {
```

```
    clog << "Algum erro estranho de processamento";
}
```

CIN, COUT, CERR E CLOG SÃO OCORRÊNCIAS DE CLASSE 818

Várias dicas anteriores mostraram como efetuar operações de E/S usando *cin*, *cout*, *cerr* e *clog*. É importante saber que esses identificadores de canal de E/S não são operadores mágicos que existem em C++. Em vez disso, *cin*, *cout*, *cerr* e *clog* são ocorrências de uma *classe de E/S*. Em dicas posteriores você aprenderá que uma *classe* define um gabarito que contém dados e métodos (funções ou operações que trabalham com os dados). Portanto, a classe é o mecanismo fundamental de C++ para a programação orientada a objetos. Ao começar a criar suas próprias classes, você poderá descansar, sabendo que está usando várias classes desde que compilou o primeiro programa. Os símbolos << e >> são simplesmente operadores de classe. Não se preocupe se esses termos parecerem confusos, pois dicas posteriores os explicarão em detalhe.

DESCARREGANDO A SAÍDA COM FLUSH 819

Como você aprendeu, seus programas podem usar o canal de E/S *cout* para exibir dados em *stdout*, e o canal de E/S *clog* para efetuar saída bufferizada em *stderr*. Quando você efetua saída bufferizada, a saída pode não aparecer na tela tão depressa quanto você deseja. Normalmente, os indicativos de arquivo e canais de E/S não descarregam a saída até que encontrem um retorno do carro ou até que ocorra uma operação de entrada. Nesses casos, seus programas podem usar *flush* para esvaziar imediatamente a saída do buffer. O programa a seguir, *flush.cpp*, ilustra como usar *flush* para descarregar os dados em *stdout* e *stderr*.

```
#include <iostream.h>

void main(void)
 {
   cout << "Isto aparece imediatamente" << flush;
   clog << "\nE isto também..." << flush;
 }
```

COMPREENDENDO O QUE IOSTREAM.H CONTÉM 820

Todos os programas apresentados até aqui incluem o arquivo de cabeçalho *iostream.h* em vez do arquivo *stdio.h*. Como você aprendeu na Dica 818, os canais de E/S *cin*, *cout*, *cerr* e *clog* são na verdade ocorrências de classe. O arquivo *iostream.h* define a classe stream correspondente e esses quatro identificadores. Não examine ainda o arquivo *iostream.h*. Você examinará o conteúdo dele mais tarde, após ter o conhecimento para compreendê-lo melhor. Por enquanto, entretanto, lembre-se apenas de que o arquivo *iostream.h* define a biblioteca de classes para E/S de tela e de teclado.

C++ REQUER PROTÓTIPOS DE FUNÇÕES 821

Na seção Funções deste livro, você aprendeu que um protótipo de função especifica o tipo de parâmetros que uma função recebe, bem como o tipo de valor que a função retorna. Quando você não especificar um protótipo de função para uma função dentro de C, o compilador gerará e exibirá uma mensagem de advertência. No entanto, em C++, você precisa especificar protótipos de funções porque, se não fizer isso, o programa não passará na compilação. O programa a seguir, *semproto.cpp*, tenta usar a função *printf* sem fornecer um tipo de função (contido no arquivo de cabeçalho *stdio.h*).

```
void main(void)
 {
   printf("Isto não passa na compilação em C++\n");
 }
```

Se você tentar compilar o programa *semproto.cpp*, C++ gerará uma mensagem de erro e a compilação terminará.

822 C++ ACRESCENTA NOVAS PALAVRAS-CHAVE

Como você aprendeu na Dica 31, uma palavra-chave é um identificador que tem significado especial para o compilador (tal como *for, while, if* e assim por diante). Além das palavras-chave que o compilador define, a Tabela 822 mostra as novas palavras-chave usadas por C++.

Tabela 822 As novas palavras-chave usadas por C++.

Palavras-chave de C++

asm	*bool*	*catch*	*class*	*delete*	*friend*	*inline*	*mutable*	*namespace*
new	*operator*	*private*	*protected*	*public*	*template*	*this*	*using*	*virtual*

823 C++ SUPORTA AS UNIÕES ANÔNIMAS

Uma união é uma estrutura de dados especial com a qual C mapeia dois ou mais membros na mesma posição de memória. Quando você declara uma união em C, precisa declarar uma variável do tipo *union*, como mostrado aqui:

```
union Valores
 {
   unsigned meus_dados;
   float  seus_dados;
 } solucao;
```

Quando você, mais tarde, quiser armazenar dados dentro da união, precisará especificar o nome da variável e o membro, como mostrado aqui:

```
solucao.meus_dados = 3;
```

No entanto, C++ permite que seus programas usem uniões anônimas (não-nomeadas). Por exemplo, o programa a seguir, *anonima.cpp*, usa uma união similar àquela que acaba de ser mostrada:

```
#include <iostream.h>

void main(void)
 {
   union
   {
     int meus_dados;
     float seus_dados;
   };

   meus_dados = 3;
   cout << "O valor de meus_dados é " << meus_dados;
   seus dados = 1.2345;
   cout << "\nO valor de seus_dados é " << seus_dados;
 }
```

Usando a união anônima, os programas podem eliminar o trabalho na programação de gerenciar um nome de união e os nomes de membros. No entanto, os nomes de membros de uma união anônima precisam ser diferentes de quaisquer outras variáveis dentro do escopo atual. Você aprenderá mais sobre as uniões anônimas em dicas posteriores.

824 RESOLVENDO O ESCOPO GLOBAL

Como você aprendeu, uma variável global é conhecida desde a sua declaração até o final do programa. Quando você usa variáveis globais, algumas vezes uma variável global tem o mesmo nome que uma variável local. Nesses casos, a função usará a variável local. No entanto, algumas vezes você irá querer referenciar a variável global dentro de uma função que tem uma variável local de nome similar. Para esses casos, C++ lhe permite preceder o nome da variável global com dois-pontos dobrados, tal como *::variável*. O programa a seguir, *global.cpp*, ilustra como usar o *operador de resolução global* de C++:

```
#include <iostream.h>

int nome_global = 1001;

void main(void)
 {
   int nome_global = 1;  // Variável local

   cout << "Valor da variável local " << nome_global << '\n';
   cout << "Valor da variável global " << ::nome_global << '\n';
 }
```

FORNECENDO VALORES PADRÃO DE PARÂMETROS 825

Parâmetros são os valores passados para as funções. A diferença principal entre os parâmetros das funções de C e C++ é que C++ permite que seus programas forneçam valores padrão para os parâmetros. Se seu programa chamar a função sem especificar um ou mais parâmetros, o programa usará os valores padrão. Por exemplo, o programa a seguir, *padrao.cpp*, usa a função *exibe_valores* para mostrar três parâmetros. Se o usuário chamar a função com menos de três parâmetros, o programa usará os valores padrão 1, 2 e 3:

```
#include <iostream.h>

void exibe_valores(int um = 1, int dois = 2, int três = 3)
 {
   cout << um << ' ' << dois << ' ' << três << '\n';
 }

void main(void)
 {
   exibe_valores(1, 2, 3);
   exibe_valores(100, 200);
   exibe_valores(1000);
   exibe_valores();
 }
```

Nota: Quando você omite parâmetros, não pode pular um parâmetro. Em outras palavras, quando omitir um parâmetro, você precisará omitir todos os parâmetros à direita do parâmetro.

CONTROLANDO A LARGURA DA SAÍDA DE COUT 826

Várias dicas nesta seção usaram o canal de E/S *cout* para exibir saída. Quando você usa *cout*, pode usar o membro width para especificar o número mínimo de caracteres usados para mostrar a saída. Por exemplo, o programa a seguir, *largura.cpp*, usa o membro *width* para selecionar uma largura mínima de cinco caracteres para a saída:

```
#include <iostream.h>

void main(void)
 {
   int i;

   for (i = 0; i < 3; i++)
    {
     cout.width(5);
     cout << i << '\n';
    }
 }
```

Quando você compilar e executar o programa *largura.cpp*, sua tela exibirá o seguinte:

```
    0
    1
```

```
   2
C:\>
```

*Nota: Quando você usa o membro **width**, precisa especificar a largura desejada para cada valor que exibir.*

827 USANDO SETW PARA DEFINIR A LARGURA PARA COUT

Na dica anterior você usou o membro *width* de *cout* para especificar o número mínimo de caracteres usados para exibir um valor. Além disso, seus programas podem usar o manipulador *setw* para especificar o texto de exibição com a largura que você quer, como mostrado aqui:

```
#include <iomanip.h>

smanip_int _Cdecl _FARFUNC setw(int _larg_desejada);
```

Por enquanto, não se preocupe em entender plenamente o protótipo da função do manipulador *setw*. O programa a seguir, *setw.cpp*, usa *setw* para selecionar larguras diferentes:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << setw(5) << 1 << '\n' << setw(6) << 2;
   cout << '\n' << setw(7) << 3;
 }
```

Quando você compilar e executar o programa *setw.cpp*, sua tela exibirá a seguinte saída:

```
    1
     2
       3
C:\>
```

Nota: Ao usar o manipulador setw, você precisará especificar a largura desejada para cada valor que exibir.

828 ESPECIFICANDO UM CARACTERE DE PREENCHIMENTO EM COUT

Por padrão, quando você usa o membro *width* de *cout* ou o manipulador *setw* para especificar caracteres de preenchimento, C++ usa o caractere de espaço para preencher os espaços adicionais. Usando o membro de preenchimento *cout*, seus programas podem especificar um caractere de preenchimento diferente. Por exemplo, o programa a seguir, *coutpts.cpp*, usa um ponto como caractere de preenchimento:

```
#include <iostream.h>
void main(void)
 {
   int i;

   for (i = 0; i < 3; i++)
    {
     cout.fill('.');
     cout.width(5 + i);
     cout << i << '\n';
    }
 }
```

Quando você compilar e executar o programa *coutapts.cpp*, sua tela exibirá a seguinte saída:

```
.....0
......1
.......2
C:\>
```

JUSTIFICANDO À DIREITA E À ESQUERDA A SAÍDA DE COUT 829

Você aprendeu que, usando o manipulador *setw* ou o membro widht de *cout*, seus programas podem especificar a largura mínima usada para exibir um valor específico. Quando você especifica uma saída, seus programas podem selecionar justificação à direita ou à esquerda usando o manipulador *setiosflags* e os membros da classe *ios*, como mostrado aqui:

```
#include <iomanip.h>

smanip_long _Cdecl _FARFUNC setiosflags(long flags);
```

Para usar *setiosflags* para selecionar a justificação à direita, coloque o seguinte manipulador no canal *cout*:

```
setiosflags(ios::right)
```

Da mesma forma, para selecionar a justificação à esquerda, use o seguinte manipulador no canal *cout*:

```
setiosflags(ios::left)
```

Por enquanto, não tente compreender o formato que esse código usa. Em vez disso, você pode usar os sinalizadores dentro do seu programa, como mostrado no próximo exemplo. Em dicas posteriores, você aprenderá como usar a classe *ios*. O programa a seguir, *diresq.cpp*, usa os sinalizadores *ios::left* e *ios::right* para selecionar a justificação à direita e à esquerda:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   int i;
   cout << "Justificação à direita\n";
   for (i = 0; i < 3; i++)
     {
       cout.width(5);
       cout << setiosflags(ios::right) << i;
     }
   cout << "\nJustificação à esquerda\n";
   for (i = 0; i < 3; i++)
     {
       cout.width(5);
       cout << setiosflags(ios::left) << i;
     }
 }
```

Quando você compilar e executar o programa *diresq.cpp*, sua tela exibirá a seguinte saída:

```
Justificação à direita
    1    2    3
Justificação à esquerda
0    1    2
C:\>
```

CONTROLANDO O NÚMERO DE DÍGITOS EM PONTO FLUTUANTE EXIBIDOS POR COUT 830

Você já sabe que o canal de E/S *cout* permite que seus programas exibam valores em ponto flutuante, como foi visto na dica 807. Quando você exibe esses valores, pode usar o manipulador *setprecision* para especificar o número desejado de dígitos à direita do ponto decimal, como mostrado aqui:

```
#include <iomanip.h>

smanip_int _Cdecl _FARFUNC setprecision(int num_digitos);
```

O programa a seguir, *defprec.cpp*, usa o manipulador *setprecision* para alterar o número de dígitos exibidos à direita do ponto decimal:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   int i;
   float valor = 1.2345;

   for (i = 0; i < 4; i++)
     cout << setprecision(i) << valor << '\n';
 }
```

Quando você compilar e executar o programa *defprec.cpp*, sua tela exibirá a seguinte saída:

```
1.2345
1.2
1.23
1.235
C:\>
```

Como você pode ver, caso especifique uma precisão de 0, *cout* exibirá todos os dígitos do valor.

831 EXIBINDO VALORES NOS FORMATOS FIXO OU CIENTÍFICO

O canal de E/S cout permite que seus programas exibam valores em ponto flutuante. Quando você usa *cout*, pode selecionar o formato fixo ou científico (exponencial) para exibir valores em ponto flutuante. Para controlar o formato de exibição do valor, seus programas podem usar os sinalizadores *ios::fixed* e *io::scientific* do manipulador *setiosflags*, como mostrado aqui:

```
#include <iomanip.h>

smanip_long _Cdecl _FARFUNC setiosflags(long flags);
```

O programa a seguir, *fixo.cpp*, usa o manipulador *setiosflags* para exibir valores nos formatos fixo e científico:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   float valor = 0.000123;

   cout << setiosflags(ios::fixed) << valor << '\n';
   cout << setiosflags(ios::scientific) << valor << '\n';
 }
```

Quando você compilar e executar o programa *fixo.cpp*, sua tela exibirá a seguinte saída:

```
0.000123
1.23e-04
C:\>
```

832 RESTAURANDO COUT PARA O PADRÃO

Várias dicas nesta seção usaram o manipulador *setiosflags* para controlar diferentes opções de formatação de *cout*. Para restaurar rapidamente as definições padrão de *cout*, você pode usar o manipulador *resetiosflags*, como mostrado aqui:

```
#include <iomanip.h>

smanip_long _Cdecl _FARFUNC resetiosflags(long flag);
```

O parâmetro *flag* especifica a opção que você quer definir. Por exemplo, o programa a seguir, *rstio.cpp*, usa o modificador para desativar a justificação à direita:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout.width(5);
   cout << setiosflags(ios::left) << 5 << '\n';
   cout.width(5);
   cout << 5 << '\n' << resetiosflags(ios::left);
   cout.width(5);
   cout << 1;
 }
```

Quando você compilar e executar o programa *rstio.cpp*, sua tela exibirá a seguinte saída:

```
5
5
    1
C:\>
```

DEFININDO A BASE DA E/S 833

Você aprendeu que, usando os modificadores *dec*, *oct* e *hex*, é possível selecionar valores decimal, octal e hexadecimal. Caso seus programas precisem exibir vários valores usando uma base específica, eles poderão usar o modificador *setbase*. O programa a seguir, *setbase.cpp*, usa o modificador *setbase* para exibir o valor 255 usando diferentes bases:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << setbase(8)  << 255 << '\n';
   cout << setbase(10) << 255 << '\n';
   cout << setbase(16) << 255 << '\n';
}
```

Quando você compilar e executar o programa *setbase.cpp*, sua tela exibirá a seguinte saída:

```
377
255
ff
C:\>
```

DECLARANDO VARIÁVEIS ONDE VOCÊ PRECISA DELAS 834

Em C, seus programas podem declarar variáveis após qualquer abre chaves. No entanto, em C++, seus programas podem declarar variáveis em qualquer posição dentro do programa. A vantagem dessas declarações é que seus programas podem declarar variáveis mais perto de onde elas serão usadas. Por exemplo, o programa a seguir, *dec_int.cpp*, declara a variável *int conta* no laço *for* dentro do qual o programa usa a variável:

```
#include <iostream.h>

void main(void)
 {
   cout << "Prestes a iniciar o laço\n";
   for (int conta = 0; conta < 10; conta++)
     cout << conta << '\n';
   cout << "Valor final de conta " << conta;
 }
```

Quando seu programa declara uma variável dentro de um bloco, o escopo da variável inicia no ponto de declaração e termina no final do bloco atual, incluindo todos os blocos que aparecem dentro do bloco atual.

835 COLOCANDO VALORES DE PARÂMETRO PADRÃO EM PROTÓTIPOS DE FUNÇÃO

Como você aprendeu, C++ lhe permite especificar valores padrão para os parâmetros da função. Normalmente, esses valores padrão aparecem no cabeçalho da função, como mostrado aqui:

```
void alguma_funcao(int a = 1, int b = 2, int c = 3)
 {
   cout << a << b << c;
 }
```

No entanto, além de colocar os valores de parâmetros padrão no cabeçalho da função, seus programas podem especificar os padrões dentro do protótipo da função, como mostrado aqui:

```
void alguma_funcao(int a = 1, int b = 2, int c = 3)

void main(void)
 {
    // Comandos
 }
```

Se a função para a qual você quer especificar parâmetros padrão não residirem no arquivo-fonte atual, você poderá instruir o compilador a incluir os padrões corretos especificando os valores no protótipo da função, como acaba de ser mostrado.

836 USANDO OPERADORES BIT A BIT E COUT

Já foi visto vimos que, para exibir dados usando o canal de E/S *cout*, seu programa usa operadores idênticos ao operador *bit a bit de deslocamento à esquerda* de C (<<). O programa a seguir, *bitscout.cpp*, usa o operador bit a bit e *cout*. Como você verá, o compilador pode determinar que operação executar pelo modo como o programa usa o operador, como mostrado aqui:

```
#include <iostream.h>

void main(void)
 {
   unsigned int valor, um = 1;

   valor = um << 1;
   cout << "Valor: " << valor << '\n';
   cout << "Resultado: " << (um << 1) << '\n';
 }
```

Quando você compilar e executar o programa *bitscout.cpp*, sua tela exibirá o seguinte:

```
Valor: 2
Resultado: 2
C:\>
```

Agrupando a operação bit a bit dentro de parênteses, o compilador associa o operador com o *operador bit a bit de deslocamento para a esquerda*. Se você removesse os parênteses, o programa produziria o seguinte resultado:

```
Valor: 2
Resultado: 11
C:\>
```

COMPREENDENDO A AVALIAÇÃO PREGUIÇOSA (OU CURTO-CIRCUITO) 837

Muitos dos programas apresentados neste livro combinam condições dentro das construções *if* e *while*, como mostrado aqui:

```
if ((a > 1) && (b < 3))
   while ((letra >= 'A') && (letra <= 'Z'))
```

Quando seus programas combinam condições dentro de construções *if* e *while*, você precisa compreender que o compilador C++ gera código que efetua avaliação *preguiçosa* ou de *curto-circuito*. A avaliação de curto-circuito significa que se o resultado de uma das condições tornar toda a condição verdadeira ou falsa, o programa deixará de efetuar o restante da condição. Por exemplo, dado o comando *if* anterior, o programa não comparará a variável *b* com o valor 3 se a primeira parte da condição *if* for falsa. Efetuar a comparação desnecessariamente consumiria tempo do processador. Da mesma forma, dado o comando *while* anterior, o programa não comparará a variável *letra* com 'Z' se *letra* não for maior ou igual a 'A'. Efetuar avaliação de curto-circuito permite que os programas poupem tempo do processador. No entanto, se você não estiver ciente do fato de que seus programas efetuam tal processamento, seus programas poderão apresentar erros. Por exemplo, considere o seguinte comando *if*:

```
if ((valor < 10) && ((letra = getchar()) != 'S'))
```

Neste caso, se a variável *valor* não for menor que 10, o programa não efetuará a segunda comparação, que usa a macro *getchar* para atribuir um valor à variável *letra*. Como resultado, algumas vezes o comando *if* atribuirá um valor a *letra* e outras vezes não atribuirá.

USANDO A PALAVRA-CHAVE CONST EM C++ 838

Como você aprendeu, a palavra-chave *const* informa o compilador de que o programa não deve alterar a variável que segue durante a execução do programa. Ao usar a palavra-chave *const* em programas C++, você poderia usar a variável correspondente de qualquer modo como normalmente usaria uma expressão constante. Por exemplo, os comandos a seguir usam a constante *tamanho_matriz* para especificar o tamanho de uma matriz string de caracteres:

```
const int tamanho_matriz = 64;

char string[tamanho_matriz];
```

A vantagem em usar constantes em relação às macros criadas com *#define* é que as constantes lhe permitem especificar informações de tipo.

USANDO A PALAVRA-CHAVE ENUM EM C++ 839

A palavra-chave *enum* permite que seus programas definam tipos enumerados. A palavra-chave *enum* em C++ é muito similar àquela que C usa, exceto, que quando você declara um tipo enumerado em C++, seu programa pode, mais tarde, usar o descritor do tipo como um tipo. Por exemplo, os comandos a seguir declaram uma variável chamada *dia* em C:

```
enum Dias { Segunda, Terca, Quarta, Quinta, Sexta };

enum Dias dia;
```

Em C++, a declaração torna-se a seguinte:

```
enum Dias { Segunda, Terca, Quarta, Quinta, Sexta };
Dias dia;
```

Como você pode ver, a segunda declaração não requer o uso da palavra-chave *enum* antes do nome de tipo *Dias*.

COMPREENDENDO O ESPAÇO LIVRE 840

Você viu anteriormente na seção Memória que seus programas podem alocar memória dinamicamente a partir do heap durante a execução. À medida que você for lendo a documentação de C++, encontrará referências ao

espaço livre. O heap e o espaço livre são a mesma coisa. Para alocar memória a partir do espaço livre, os programas C++ usam *new* e *delete*. É importante observar que, ao contrário de *malloc* e *free*, que são funções, *new* e *delete* são operadores. A dica a seguir mostra como alocar memória com *new*.

841 ALOCANDO MEMÓRIA COM NEW

Como você aprendeu, os programas C++ alocam memória dinâmica a partir do espaço livre usando o operador *new*. Para usar *new*, um programa precisa especificar o número desejado de bytes. O programa a seguir, *novamatz.cpp*, aloca memória para uma matriz de 256 bytes. O programa então preenche a matriz com a letra A e exibe o conteúdo da matriz:

```
#include <iostream.h>

void main(void)
 {
   char *matriz = new char[256];
   int i;

   for (i = 0; i < 256; i++)
      matriz[i] = 'A';
   for (i = 0; i < 256; i++)
      cout << matriz[i] << ' ';
 }
```

842 ALOCANDO MÚLTIPLAS MATRIZES

Na dica anterior, você usou o operador *new* para alocar dinamicamente uma string de caracteres de 256 bytes. Naquele momento, o compilador alocou a memória quando o programa declarou a variável ponteiro, como mostrado aqui:

```
char *matriz = new char[256];
```

No entanto, quando seus programas usam o operador *new* para alocar memória, eles podem fazer isso a partir de qualquer posição. O programa a seguir, *new_copi.cpp*, usa o operador *new* para alocar três strings de caracteres — cada uma em uma posição diferente no programa:

```
#include <iostream.h>

void main(void)
 {
   char *matriz = new char[256];
   char *alvo, *destino;
   int i;

   alvo = new char[256];
   for (i = 0; i < 256; i++)
    {
     matriz[i] = 'A';
     alvo[i] = 'B';
    }

   destino = new char[256];
   for (i = 0; i < 256; i++)
    {
     destino[i] = alvo[i];
     cout << destino[i] << ' ';
    }
 }
```

TESTANDO A FALTA DE ESPAÇO LIVRE 843

A seção Gerenciamento de Memória no Windows que, quando o heap não pode satisfazer uma solicitação, as funções *calloc* e *malloc* retornam *NULL*. O mesmo é verdadeiro para o operador *new* e o espaço livre. O programa a seguir, *semlivre.cpp*, usa o operador *new* para alocar memória até que o espaço livre esteja vazio:

```
#include <iostream.h>

void main(void)
 {
   char *ponteiro;

   do
   {
     ponteiro = new char[10000];
     if (ponteiro)
        cout << "10.000 bytes alocados\n";
     else
        cout << "A alocação falhou\n";
   } while (ponteiro);
 }
```

CONSIDERAÇÕES SOBRE O ESPAÇO NO HEAP 844

C++ refere-se ao heap como o espaço livre. Dependendo do seu modelo de memória, a quantidade de espaço de heap disponível diferirá. Por exemplo, você poderia compilar o programa *semlivre.cpp* usando o modelo de memória large. Se você fizer isso, o programa *semlivre.cpp* compilado com o modelo de memória large executará de forma mais demorada e alocará consideravelmente mais espaço no heap antes de falhar.

USANDO PONTEIROS FAR E O OPERADOR NEW 845

Como você aprendeu, o operador *new* permite que seus programas aloquem memória a partir do espaço livre. Se você estiver usando o modelo de memória small, o espaço livre corresponderá ao heap *NEAR*. Caso seus programas precisem alocar memória que o heap near fornece, seus programas poderão alocar ponteiros far. O programa a seguir, *novo_far.cpp*, aloca ponteiros far a partir do espaço livre até que o heap far fique sem memória:

```
#include <iostream.h>

void main(void)
 {
   char far *ponteiro;

   do
   {
     ponteiro = new far char[10000];
     if (ponteiro)
        cout << "10.000 bytes alocados\n";
     else
        cout << "A alocação falhou\n";
   } while (ponteiro);
 }
```

LIBERANDO MEMÓRIA DE VOLTA AO ESPAÇO LIVRE 846

Quando seus programas C++ alocam memória dinamicamente, seus programas devem liberar a memória assim que não precisarem mais dela. Quando seus programas C alocam memória usando *calloc* e *malloc* devem, mais tarde, liberar a memória usando *free*. Quando seus programas C++ alocam memória usando *new*, devem, mais tarde, liberar a memória usando *delete*. O programa a seguir, *delete.cpp*, usa o operador *delete* para liberar três matrizes alocadas dinamicamente de volta para o espaço livre:

```
#include <iostream.h>

void main(void)
 {
   char *matriz = new char[256];
   char *alvo, *destino;
   int i;

   alvo = new char[256];
   for (i = 0; i < 256; i++)
    {
      matriz[i] = 'A';
      alvo[i] = 'B';
    }
   delete matriz;
   destino = new char[256];
   for (i = 0; i < 256; i++)
    {
      destino[i] = alvo[i];
      cout << destino[i] << ' ';
    }
   delete alvo;
   delete destino;
 }
```

847 COMPREENDENDO AS REFERÊNCIAS DE C++

Um *apelido* é um segundo nome para uma variável. Em C, seus programas podem criar apelidos usando ponteiros. C++ simplifica a criação de apelidos usando referências. Para criar uma referência, você usa o operador de referência (&), como mostrado aqui:

```
int variavel;
int& apelido = variavel;
```

O operador de referência é similar ao operador de endereço de C. No entanto, observe a posição do operador. O operador de referência segue imediatamente um tipo (tal como *int*, *float* ou *char*). Cada apelido pode corresponder a somente uma variável por toda a sua duração. O programa a seguir, *apelido.cpp*, cria dois apelidos e usa-os para exibir os endereços de variáveis específicas:

```
#include <iostream.h>

void main(void)
 {
   int a = 1001;
   int& um_apelido = a;
   float preco = 39.95;
   float& preco_apelido = preco;

   cout << "O valor de a é " << a << " O apelido é " << um_apelido;
   cout << "\nO preco é " << preco << " O apelido é " << preco_apelido;
   um_apelido++;
   cout << "\nO valor de a é " << a << " O apelido é " << um_apelido;
 }
```

O programa *apelidos.cpp* usa a variável de referência *um_apelido* para incrementar o valor de *a*. Quando um programa refere-se a uma referência, quaisquer operações corresponderão diretamente à variável apelidada.

PASSANDO UMA REFERÊNCIA A UMA FUNÇÃO 848

Como você aprendeu, para alterar uma variável dentro de uma função, seus programas precisam passar um ponteiro para a variável. Quando você usa C++, pode simplificar a alteração de uma variável dentro de uma função usando uma *referência*. Usando uma referência, você elimina a necessidade de um operador de ponteiro (->). O programa a seguir, *funcref.cpp*, passa uma referência à variável *valor* para a função *altera_valor*, que atribui o valor 1500 à variáve.:

```
#include <iostream.h>

void muda_valor(int& valor_referencia)
 {
   valor_referencia = 1500;
 }

void main(void)
 {
   int valor = 10;
   int& apelido = valor;

   cout << 'Valor antes da função: " << valor << '\n';
   muda_valor(apelido);
   cout << 'Valor após a função: " << valor << '\n';
 }
```

Como você pode ver, usar uma referência simplifica o processo de alteração de um valor dentro de uma função.

ATENTANDO PARA OS OBJETOS OCULTOS 849

Uma referência cria um segundo nome para uma variável — um apelido. Quando você cria referências, precisa garantir que o tipo de referência seja idêntico ao tipo que ele referenciará. Por exemplo, o comando a seguir cria uma referência a uma variável do tipo *int*:

```
int valor;
int& apelido = valor;
```

Se um tipo de referência e tipo de variável diferem, C++ cria um *objeto oculto* que não apelida o valor especificado, mas, em vez disso, contém o valor para uma variável não-nomeada do tipo de referência. Por exemplo, o comando a seguir cria um objeto oculto do tipo *float*:

```
int valor;
float& apelido = valor;
```

Como você pode ver, a referência e os tipos de variável diferem. Portanto, o compilador não apelidará a variável *valor*, mas, em vez disso, alocará memória para um valor em ponto flutuante, apelidando a memória com a referência dada. O porquê de você precisar estar ciente de objetos ocultos é que elas podem levar a erros que são difíceis de detectar. Se você modificar o tipo de uma variável, certifique-se de também alterar o tipo da referência correspondente, se uma referência existir.

USANDO TRÊS MODOS DE PASSAR PARÂMETROS 850

Em C, seus programas podem passar parâmetros para as funções usando *chamada por valor* ou *chamada por referência de ponteiro*. Como você aprendeu, em C++ seus programas podem usar uma terceira técnica, a *chamada por referência*. O programa a seguir, *chama_3.cpp*, ilustra como usar todas as três técnicas de chamada:

```
#include <iostream.h>
#include <iomanip.h>

void chama_por_valor(int a, int b, int c)
```

```
{
  a = 3; b = 2; c = 1;
}

void chama_por_ponteiro_referencia(int *a, int *b, int *c)
{
  *a = 3; *b = 2; *c = 1;
}

void chama_por_referencia(int& a, int& b, int& c)
{
  a = 1; b = 2; c = 3;
}

void main(void)
{
  int a = 1, b = 2, c = 3;
  int& a_apelido = a;
  int& b_apelido = b;
  int& c_apelido = c;

  chama_por_valor(a, b, c);
  cout << "Por valor: " << a << b << c << '\n';
  chama_por_ponteiro_referencia(&a, &b, &c);
  cout << "Por ponteiro: " << a << b << c << '\n';
  chama_por_referencia(a_apelido, b_apelido, c_apelido);
  cout << "Por referência: " << a << b << c << '\n';
}
```

851 REGRAS PARA TRABALHAR COM REFERÊNCIAS

Em C++, uma referência lhe permite criar um apelido para uma variável. Quando você usar referências, lembre-se das seguintes regras:

1. *Após a inicialização, seu programa não pode alterar um valor de referência.*
2. O tipo de referência e o tipo de variável precisam ser os mesmos.
3. Você não pode criar um ponteiro para uma referência.
4. Você não pode comparar o valor de duas referências — as comparações poderão comparar os valores das variáveis referenciadas.
5. Você não pode incrementar, decrementar ou alterar um valor de referência — as operações se aplicarão ao valor da variável referenciada.
6. Você pode distinguir o operador de referência a partir do operador de endereço porque o operador de referência sempre segue o tipo (por exemplo, *int&*).

852 AS FUNÇÕES PODEM RETORNAR REFERÊNCIAS

Em C++, uma referência é um apelido para uma variável. Como você aprendeu, as referências podem simplificar a passagem de parâmetro eliminando a necessidade de efetuar operações de ponteiro. Todas as dicas apresentadas até aqui inicializarão as variáveis de referência na declaração, perto do início de *main*. No entanto, C++ também permite que as funções retornem referências. Como C++ permite que seus programas declarem variáveis em qualquer posição, seus programas podem, portanto, criar e inicializar uma referência em qualquer posição no seu programa retornando uma referência a partir de uma função. O programa a seguir, *rtn_ref.cpp*, chama a função *le_livro*, que retorna uma referência para uma variável do tipo *livro*:

```
#include <iostream.h>

struct livro
{
  char autor[64];
  char titulo[64];
```

```
    float preco;
};

livro biblio[3] = {
  { "Jamsa e Klander", "Bíblia do Programador C/C++", 49.95 },
  { "Klander", "Hacker Proof", 54.95 },
  { "Jamsa e Klander", "1001 Dicas Sobre Visual Basic", 54.95 }};

livro& pega_livro(int i)
{
   if ((i >= 0) && (i < 3))
      return(biblio[i]);
   else
      return(biblio[0]);
}

void main(void)
{
   cout << "Prestes a pegar livro 0\n";
   livro& este_livro = pega_livro(0);
   cout << este_livro.autor << ' ' << este_livro.titulo;
   cout << ' ' << este_livro.preco;
}
```

USANDO A PALAVRA-CHAVE INLINE DE C++ 853

Você já sabe que os programas passam parâmetros para as funções usando a pilha. Cada vez que seu programa chama uma função, o computador precisa colocar os parâmetros da função (e o endereço de retorno do programa) na pilha, e, depois, restaurar esses valores. Essas operações push e pop levam a uma sobrecarga que torna o uso das funções ligeiramente mais lento do que inserir o código das funções diretamente no programa. Se seus programas têm uma ou duas funções críticas que precisam executar rapidamente, você deve usar a palavra-chave *inline* para instruir o compilador a colocar o código in-line correspondente no programa em cada chamada de função, em vez de criar código de função separado. Se seus programas chama a função in-line a partir de cinco posições diferentes, o compilador inserirá a função correspondente no programa cinco vezes. Se seus programas chamam a função in-line a partir de 50 posições, o compilador inserirá o código 50 vezes. Portanto, o código in-line envolve uma escolha entre tempo e espaço. Usar o código in-line cria um programa mais rápido, mas também torna o código do programa maior (o que, em teoria, pode tornar o programa mais lento). O programa a seguir, *inline cpp*, usa duas funções similares, colocando uma função in-line e chamando a segunda função. O programa exibe a quantidade de tempo necessária para chamar cada função 30.000 vezes:

```
#include <iostream.h>
#include <time.h>

inline void permuta_inline(int *a, int *b, int *c, int *d)
{
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
   temp = *c;
   *c = *d;
   *d = temp;
}

void permuta_chamada(int *a, int *b, int *c, int *d)
{
   int temp;
```

```
    temp = *a;
    *a = *b;
    *b = temp;
    temp = *c;
    *c = *d;
    *d = temp;
  }

void main(void)
  {
    clock_t inicio, parar;
    long int i;

    int a = 1, b = 2, c = 3, d = 4;
    inicio = clock();
    for (i = 0; i < 300000L; i++)
       permuta_inline(&a, &b, &c, &d);
    parar = clock();
    cout << "Tempo para inline: " << parar - inicio;
    inicio = clock();
    for (i = 0; i < 300000L; i++)
       permuta_chamada(&a, &b, &c, &d);
    parar = clock();
    cout << "\nTempo para a função chamada: " << parar - inicio;
  }
```

854 USANDO A PALAVRA-CHAVE ASM DE C++

Como você aprendeu, dependendo do propósito do seu programa, algumas vezes você precisa efetuar programação em linguagem assembly. Nesses casos, você pode criar uma função de linguagem assembly e ligar a função ao seu programa, ou pode usar a palavra-chave *asm* para inserir comandos em linguagem assembly no seu código em C++. O programa a seguir, *asm_demo.cpp*, usa a palavra-chave *asm* para incluir os comandos em linguagem Assembly necessários para soar o alto-falante interno do computador:

```
#include <iostream.h>

void main(void)
  {
    cout << "Prestes a soar o alto-falante...\n";
    asm
    {
       MOV AX,0x0200
       MOV DL,7
       INT 0x21
    };
    cout << "Acabado..\n";
  }
```

855 LENDO UM CARACTERE USANDO CIN

Várias dicas anteriores usaram o canal de E/S *cin* para ler a entrada a partir do teclado. Para aumentar seu controle sobre o teclado e a entrada redirecionada, seus programas podem usar *cin.get* para ler a entrada de um caractere de cada vez, como mostrado aqui:

```
caractere = cin.get();
```

O programa a seguir, *cin_get.cpp*, usa *cin.get* para atribuir caracteres até, mas não incluindo, o caractere de nova linha para a string de caracteres *str*:

```
#include <iostream.h>
```

```
#include <stdio.h>

void main(void)
 {
   char str[256];
   int i = 0;

   while ((str[i] = cin.get()) != '\n')
      i++;
   str[i] = NULL;
   cout << "A string era: " <<  str;
 }
```

ESCREVENDO UM CARACTERE COM COUT 856

Na dica anterior você aprendeu que seus programas podem receber caracteres um de cada vez usando *cin.get*. De um modo similar, seus programas podem usar *cout.put* para escrever um caractere, como mostrado aqui:

```
cout.put(caractere);
```

O programa a seguir, *cout_put.cpp*, usa *cout* para escrever uma string de caractere um caractere de cada vez:

```
#include <iostream.h>

void main(void)
{
   char *titulo = "Bíblia do Programador C/C++, do Jamsa!";
   while (*titulo)
      cout.put(*titulo++);
}
```

ESCREVENDO UM PROGRAMA SIMPLES DE FILTRO 857

Como você aprendeu, *cout.put* e *cin.get* permitem que seus programas efetuem E/S de caracteres. O programa a seguir, *maiusc.cpp*, converte a entrada redirecionada para maiúsculas. Para efetuar a conversão, o programa simplesmente repete um laço até que *cin get* retorne -1, indicando o final do arquivo.

```
#include <iostream.h>
#include <ctype.h>

void main(void)
{
   char letra;

   while ((letra = cin.get()) != -1)
     cout.put(toupper(letra));
}
```

ESCREVENDO UM COMANDO TEE SIMPLES 858

C++ lhe permite redirecionar a saída do canal de E/S *cout*. O programa a seguir, *tee.cpp*, escreve sua entrada redirecionada nos canais de E/S *cout* e *cerr*. Como o programa usa dois canais de E/S, você pode visualizar a entrada do programa na tela e ainda redirecionar a saída para uma fonte diferente:

```
#include <iostream.h>

void main(void)
 {
   char letra;

   while ((letra = cin.get()) != -1)
    {
       cout.put(letra);
       cerr.put(letra);
    }
 }
```

859 ESCREVENDO UM COMANDO FIRST SIMPLES

Os canais de E/S *cin* e *cout* suportam o redirecionamento da E/S. O programa a seguir, *simples1.cpp*, usa esses canais de entrada para escrever as primeiras dez linhas da entrada redirecionada na tela:

```
#include <iostream.h>

void main(void)
 {
   char letra;
   int conta = 0;

   while ((letra = cin.get()) != -1)
    {
       cout.put(letra);
       if ((letra == '\n') && (++conta == 10))
          break;
    }
 }
```

860 ESCREVENDO UM COMANDO FIRST MELHOR

Na dica anterior você criou o programa *simples1.cpp*, que exibiu as dez primeiras linhas da entrada redirecionada. Um comando mais flexível permitiria que o usuário especificasse, como um argumento de linha de comando, o número de linhas que o usuário quer exibir. O programa a seguir, *primeiro.cpp*, permite que o usuário faça exatamente isso:

```
#include <iostream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   char letra;
   int conta = 0;
   int limite_linha;

   limite_linha = atoi(argv[1]);
   while ((letra = cin.get()) != -1)
    {
       cout.put(letra);
       if ((letra == '\n') && (++conta == limite_linha))
          break;
    }
 }
```

Se o usuário não especificar o número de linhas para o programa exibir, ou se o usuário especificar uma quantidade inválida de linhas, o programa exibirá toda a entrada redirecionada.

TESTANDO O FINAL DO ARQUIVO 861

Várias dicas anteriores usaram *cin.get* para determinar o final da entrada redirecionada, como mostrado aqui:

```
while ((letra = cin.get()) != -1)
```

Além de testar o valor de retorno do método *cin.get*, seus programas podem testar *cin.eof*, como segue:

```
while (! cin.eof())
```

O programa a seguir, *prim_eof.cpp*, modifica o programa *primeiro.cpp* para testar o final de arquivo com o método *cin.eof*:

```
#include <iostream.h>
#include <stdlib.h>

void main(int argc, char **argv)
 {
   char letra;
   int conta = 0;
   int limite_linha;

   limite_linha = atoi(argv[1]);
   while (! cin.eof())
    {
      letra = cin.get();
      cout.put(letra);
      if ((letra == '\n') && (++conta == limite_linha))
         break;
    }
 }
```

GERANDO UMA NOVA LINHA COM ENDL 862

Muitas dicas anteriores colocaram o caractere de nova linha (\n) no canal de saída *cout* para gerar um retorno do carro e alimentação de linha. Além de usar o caractere de nova linha, seus programas podem usar *endl*, como mostrado aqui:

```
cout << "Olá, pessoal!" << endl;
```

O programa a seguir, *endl.cpp*, usa *endl* várias vezes para gerar um retorno do carro e alimentação de linha:

```
#include <iostream.h>

void main(void)
 {
   cout << "Esta é a linha 1" << endl;
   cout << "Esta é a linha 2" << endl;
   cout << "Esta é a linha 3--";
   cout << "A última linha" << endl;
 }
```

Quando você compilar e executar o programa *endl.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a linha 1
Esta é a linha 2
Esta é a linha 3--É a última linha
C:\>
```

863 COMPREENDENDO AS ESPECIFICAÇÕES DE LIGAÇÃO

Como você aprendeu, C++ requer protótipos de função para cada função que seu programa usar. O compilador C++ usa os protótipos para verificar os tipos dos parâmetros e os valores de retorno. Durante a compilação, o compilador altera os nomes das funções e seus parâmetros no código-objeto resultante. O linkeditor, por sua vez, usa esses novos nomes para resolver as referências externas. Infelizmente, se você estiver vinculando o código para o qual um compilador C compilou anteriormente, os nomes das funções no código-objeto não estarão no mesmo "formato de nome de função C++". Para evitar que o compilador C++ altere os nomes das funções C, você poderá usar um *especificador de* linkage. Em resumo, o especificador de *linkage* informa o compilador C++ do formato correto que ele deverá usar para nomear as funções no arquivo-objeto. Por exemplo, assuma que você tenha uma função chamada *calcula_folha* que você (ou outro programador) escreveu anteriormente em C. Para instruir o compilador C++ a não mudar o formato do nome da função, você usaria o seguinte especificador de linkage:

```
extern "C"
 {
   float calcula_folha(int func_conta, char *func_arq);
 };
```

Nota: Se você examinar os arquivos de cabeçalho que seu compilador fornece, encontrará vários especificadores de linkage similares ao mostrado nesta dica.

864 COMPREENDENDO A SOBRECARGA

Sobrecarga é o processo de atribuir mais de uma operação a um operador ou fornecer duas ou mais funções com o mesmo nome. Por exemplo, C e C++ usam o sinal de adição (+) como o operador de *adição*. Você pode usar a sobrecarga para instruir C++ a também usar o símbolo de adição para concatenar as strings, como mostrado aqui:

```
nomecaminho = nome_diretorio + nomearq;
```

Dependendo de como seu programa usa o sinal de adição, o compilador C++ determinará se o comando do programa efetuará adição ou concatenação de strings. Você também aprendeu que, quando usa C, algumas vezes é preciso criar funções nomeadas de forma diferente que trabalham com valores de tipos diferentes. Por exemplo, se você criou uma função que retorna a soma dos valores em uma matriz de inteiros, precisa criar uma função com um nome diferente — se quiser somar os valores em uma matriz do tipo *float*. Como você aprenderá, C++ lhe permite sobrecarregar as funções e os operadores, o que simplifica muitas operações.

865 SOBRECARREGANDO AS FUNÇÕES

C++ lhe permite ter múltiplas funções com o mesmo nome. Durante a compilação, o compilador C++ determina qual função chamar, com base no número e tipos de parâmetros que o comando de chamada passa para a função. Por exemplo, o programa a seguir, *sobrecar.cpp*, cria duas funções chamadas *soma* que retornam a soma do número de elementos em uma matriz. A primeira função suporta matrizes do tipo *float*, enquanto a segunda suporta matrizes do tipo *int*:

```
#include <iostream.h>

int soma(int *matriz, int elemento_conta)
 {
   int result = 0;
   int conta;

   for (conta = 0; conta < elemento_conta; conta++)
      result += matriz[conta];
   return(result);
 }

float soma(float *matriz, int elemento_conta)
 {
   float result = 0;
```

```
   int conta;

   for (conta = 0; conta < elemento conta; conta++)
      result += matriz[conta];
   return(result);
 }

void main(void)
 {
   int a[5] = { 1, 2, 3, 4, 5 };
   float b[4] = { 1.11, 2.22, 3.33, 4.44 };

   cout << "Soma dos valores int: " << soma(a, 5) << '\n';
   cout << "Soma dos valores float: " << soma(b, 4) << '\n';
 }
```

SOBRECARREGANDO FUNÇÕES: UM SEGUNDO EXEMPLO 866

Como você aprendeu, C++ lhe permite sobrecarregar as funções, criando duas ou mais funções nos seus programas que têm o mesmo nome. O programa a seguir sobrecarrega a função *permuta*. A primeira função permuta dois valores, enquanto a segunda função permuta quatro. Durante a compilação, o compilador usa o número de parâmetros para determinar qual função chamar. *Usaparam.cpp* é seu primeiro programa com função sobrecarregada, como mostrado aqui:

```
#include <iostream.h>

void permuta(int *a, int *b)
 {
   int temp = *a;
   *a = *b;
   *b = temp;
 }

void permuta(int *a, int *b, int *c, int *d)
 {
   int temp = *a;
   *a = *b;
   *b = temp;
   temp = *c;
   *c = *d;
   *d = temp;
 }

void main(void)
 {
   int a = 1, b = 2 , c = 3, d = 4;

   permuta(&a, &b);
   cout << "Permutei a e b " << a << b << '\n';
   permuta(&a, &b, &c, &d);
   cout << "Permutei quatro " << a << b << c << d << '\n';
 }
```

EVITANDO A AMBIGÜIDADE DA SOBRECARGA 867

Quando você cria funções sobrecarregadas, é possível criar uma situação na qual o compilador não consegue distinguir entre duas (ou mais) funções sobrecarregadas. Quando você cria duas ou mais funções sobrecarregadas, entre as quais o compilador não consegue distinguir, o compilador considera as funções *ambíguas*. As chamadas de funções ambíguas são erros, e o compilador não compilará seu programa.

De longe, a causa mais comum de ambigüidade envolve as conversões automáticas de tipo de C++. Como você aprendeu, C++ automaticamente tenta converter os argumentos que o programa usa para chamar uma função no tipo de argumentos que a função espera. Por exemplo, considere o seguinte fragmento de código:

```
int minhafunc(double d);
//
// Código do programa aqui
//
cout << minhafunc('c');     // C++ converte para inteiro
```

Como indica o comentário dentro do fragmento de código, a chamada da função no exemplo não causa um erro porque C++ automaticamente converte o caractere para seu equivalente double. C++ recusa-se a reconhecer poucas conversões de tipo da forma mostrada no exemplo anterior. Embora as conversões de tipo sejam convenientes, elas podem causar sérios problemas quando você sobrecarrega as funções. O programa a seguir, *sobr_err.cpp*, sobrecarrega a função *func_exemplo* com dois tipos de parâmetros diferentes: *float* e *double*. O programa chama a função duas vezes — uma vez com o valor 1500.1, que é um tipo de parâmetro *double* e, portanto, não causa ambigüidade; e uma vez com o valor 1500, que causa ambigüidade pois o compilador não sabe se deve converter o valor para um *float* ou um *double*. Quando você compilar o programa a seguir, receberá uma mensagem de erro do compilador:

```
#include <iostream.h>

float funcao_exemplo(float i);
double funcao_exemplo(double i);

void main(void)
 {
   cout << funcao_exemplo(1500.1) << " "; // não-ambíguo, chama funcao_
   exemplo(double)
   cout << funcao_exemplo(1500);    // ambíguo
 }

float funcao_exemplo(float i)
 {
   return i;
 }

double funcao_exemplo(double i)
 {
   return -i;
 }
```

Nota: Na Dica 814 você aprendeu a não passar uma referência para cin. Como você aprendeu nesta dica, passar uma referência para uma função sobrecarregada confundirá o compilador. Como pode processar múltiplos tipos de valores, cin também precisa ser uma função sobrecarregada — o que explica por que você não pode usar referências com ela.

868 LENDO UMA LINHA DE CADA VEZ COM CIN

Como você aprendeu, seus programas podem ler a entrada do teclado usando *cin*. Quando seus programas querem ler a entrada um caractere de cada vez, eles podem usar *cin.get*. Em alguns casos, seus programas podem precisar efetuar operações de entrada uma linha de cada vez. Para esses casos, eles podem usar *cin.getline*, como mostrado aqui:

```
char string[256];

cin.getline(string, sizeof(string), '\n');
```

O parâmetro *string* é um ponteiro para a string que você quer que *cin.getline* leia. O operador *sizeof* especifica o número de bytes que a string pode armazenar. Finalmente, o caractere de nova linha especifica o caractere que terminará a leitura. O programa a seguir, *getline.cpp*, usa *cin.getline* para ler uma linha de entrada:

```
#include <iostream.h>

void main(void)
 {
   char string[256];

   cout << "Digite seu nome completo e pressione Enter\n";
   cin.getline(string, sizeof(string), '\n');
   cout << string;
 }
```

Para experimentar com *cin.getline*, altere o caractere de finalização (por exemplo) para uma letra do alfabeto, e, depois, observe os resultados do programa.

USANDO CIN.GETLINE EM UM LAÇO 869

Na Dica 857, seus programas usaram *cin.get* e *cout.put* para exibir a entrada redirecionada como maiúsculas. O programa a seguir, *tudomais.cpp*, usa *cin.getline* e *cout* para efetuar processamento similar:

```
#include <iostream.h>
#include <string.h>

void main(void)
 {
   char string[256];

   while (cin.getline(string, sizeof(string), '\n'))
     cout << strupr(string) << '\n';
 }
```

Como você pode ver, o programa repete um laço até que *cin.getline* retorne 0, o que indica o final da entrada redirecionada. Como a função *cin.getline* não coloca o caractere de nova linha na string, *cout* precisa escrever o caractere de nova linha para cada linha.

ALTERANDO O TRATAMENTO NORMAL DO OPERADOR NEW 870

Quando o operador *new* não puder alocar memória suficiente para satisfazer uma solicitação de memória, ele retornará *NULL*. Dependendo da função do seu programa, algumas vezes você irá querer que *new* efetue outro processamento quando ele não puder alocar memória. Quando *new* não puder alocar memória, ele poderá chamar a função apontada por um ponteiro global para uma função chamada *_new_handler*. Atribuindo a variável *_new_handler* para apontar para uma função personalizada, você poderá instruir *new* a chamar sua própria função quando ele não puder alocar memória. O programa a seguir, *new_rot.cpp*, usará o ponteiro da função *_new_handler* para instruir *new* a chamar a função *sem_memória* quando não puder satisfazer uma solicitação de alocação de memória:

```
#include <iostream.h>
#include <stdlib.h>

extern void (*_new_handler)();

void sem_memoria(void)
 {
   cerr << "Não há mais memória para alocar...\n";
   exit(0);
 }

void main(void)
 {
   _new_handler = sem_memoria;
   char *ptr;
   do
```

```
    {
      ptr = new char[10000];
      if (ptr)
        cout << "Acabo de alocar 10.000 bytes\n";
    } while (ptr);
  }
```

Nota: Caso sua rotina de tratamento não puder alocar a memória para o programa, a rotina de tratamento precisará finalizar o programa, ou um laço infinito ocorrerá.

871 DEFININDO UMA NOVA ROTINA DE TRATAMENTO COM SET_NEW_HANDLER

Na dica anterior, você aprendeu que C++ lhe permite definir sua própria rotina de tratamento que seus programas chamarao quando o operador *new* não puder satisfazer uma solicitação de memória. Para atribuir a nova rotina de tratamento, seu programa atribui o endereço da rotina de tratamento da sua função para uma variável global chamada *_new_handler*. Para simplificar o processo de atribuição da nova rotina de tratamento, muitos compiladores C++ fornecem uma função chamada *set_new_handler*, como mostrado aqui:

```
#include <new.h>

void (* set_new_handler(void (* rotina_personal)()))();
```

O programa a seguir, *def nova.cpp*, usa a função *set_new_handler* para instalar uma rotina de tratamento personalizada:

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void nenhuma_memoria(void)
 {
   cerr << "Não há mais memória para alocar...\n";
   exit(0);
 }

void main(void)
 {
   char *ptr;

   set_new_handler(nenhuma_memoria);
   do
   {
     ptr = new char[10000];
     if (ptr)
       cout << "Acabo de alocar 10.000 bytes\n";
   } while (ptr);
 }
```

872 DETERMINANDO UMA COMPILAÇÃO C++

Muitos compiladores C++ lhe permitem compilar programas C padrão. Dependendo dos comandos do seu programa, algumas vezes você quer terminar uma compilação C padrão. Quando muitos compiladores C++ compilam um programa C++, eles poderiam definir uma constante que você pode testar dentro de seus programas. Por exemplo, o compilador *Turbo C++ Lite* define a constante *__cplusplus* quando está compilando um programa C++. O programa a seguir, *testacpp.cpp*, usa a constante *__cplusplus* para determinar se o compilador está efetuando uma compilação C ou C++:

```
#ifdef __cplusplus
#include <iostream.h>
```

```
#else
#include <stdio.h>
#endif

void main(void)
 {
#ifdef __cplusplus
   cout << "Compilação de C++ ";
#else
   printf("Compilação de C \n");
#endif
 }
```

Compile o programa *testacpp.cpp* como um arquivo com a extensão CPP. Em seguida, copie o conteúdo para um arquivo com a extensão C e compile o arquivo. Observe o processamento que o compilador efetua.

COMPREENDENDO AS ESTRUTURAS EM C++ 873

Você já sabe que uma estrutura lhe permite agrupar informações relacionadas de diferentes tipos. Ao declarar uma estrutura em C, é possível especificar um nome com o qual, mais tarde, você possa declarar variáveis do tipo estrutura, como mostrado aqui:

```
struct nome
 {
   int membro_a;
   float membro_b;
   char membro_c[256];
 };
struct nome variavel_um, variavel_dois;
```

No entanto, quando você declarar uma estrutura em C++, o nome da estrutura se tornará um tipo, com o qual seu programa pode, mais tarde, declarar variáveis sem especificar a palavra-chave *struct*, como mostrado aqui:

```
struct nome
 {
   int membro_a;
   float membro_b;
   char membro_c[256];
 };
nome variavel_um, variavel_dois;
```

Como você pode ver, a estrutura C++ não requer a palavra-chave *struct* diante do nome na declaração da variável.

INTRODUZINDO FUNÇÕES COMO MEMBROS DA ESTRUTURA 874

Quando você cria programas C, o compilador C lhe permite usar ponteiros para funções como membros de estrutura, como mostrado aqui:

```
struct nome
 {
  int membro_a;
  int *membro_b)();//Ponteiro para função que retorna int
 };
```

C++ lhe permite levar o conceito anterior um passo adiante, lhe permitindo colocar funções reais dentro de uma estrutura, como mostrado aqui:

```
struct nome
 {
   int membro_a;
   int membro_b();
 };
```

Ao declarar a função correspondente, você tem duas escolhas. Como mostrará a dica a seguir, você pode definir o código da função imediatamente dentro da estrutura, ou pode definir o código da função fora da estrutura, como mostrará a dica 876. Para chamar a função, seu programa simplesmente referencia o membro da estrutura, como mostrado aqui:

```
variavel.membro_b(parametros);
```

875 DEFININDO UMA FUNÇÃO-MEMBRO DENTRO DE UMA ESTRUTURA

Como você aprendeu, C++ permite que seus programas coloquem funções como membros de estrutura. Quando sua estrutura contém um membro que é uma função, você pode definir o código da função correspondente dentro da estrutura. O programa a seguir, *func_mbr.cpp*, define a função que corresponde ao membro *exib_msg*:

```
#include <iostream.h>

struct Msg
 {
   char mensagem[256];
   void exibe_mensagem(void) { cout << mensagem; }
 };

void main(void)
 {
   struct Msg livro = { "Bíblia do Programador C/C++" };

   livro.exibe_mensagem();
 }
```

O programa *func_mbr.cpp* chama a função-membro *exibe_mensagem*, que, por sua vez, exibe o membro *mensagem*. Como mostra o código anterior, *func_mbr.cpp* define *exibe_mensa* dentro da estrutura. Na dica a seguir, você aprenderá como definir a função fora da estrutura.

876 DECLARANDO UMA FUNÇÃO-MEMBRO FORA DE UMA ESTRUTURA

Você já viu que C++ lhe permite colocar funções como membros em uma estrutura. Na dica anterior, você definiu a função-membro *exibe_mensagem* dentro da própria estrutura. O programa a seguir, *func_2.cpp*, define a função fora da estrutura. Para corresponder a função com a estrutura *Msg*, o programa precede o nome da função com o nome da estrutura, seguida por dois-pontos dobrados:

```
#include <iostream.h>

struct Msg
 {
   char mensagem[256];
   void exibe_mensagem(char *mensagem);
 };

void Msg::exibe_mensagem(char *mensagem)
 {
   cout << mensagem;
 }

void main(void)
 {
   struct Msg livro = { "Bíblia do Programador C/C++" };
   livro.exibe_mensagem(livro.mensagem);
 }
```

PASSANDO PARÂMETROS PARA UMA FUNÇÃO-MEMBRO 877

Como já visto, C++ lhe permite colocar funções como um membro de uma estrutura. As Dicas 875 e 876 usaram a função-membro *exibe_mensagem* dentro de uma estrutura *Msg*. Quando você coloca uma função como um membro de estrutura, pode tratar a função exatamente como faria com qualquer outra função C++. Em outras palavras, você pode passar parâmetros para a função e declarar variáveis locais dentro da função. O programa a seguir, *func_3.cpp*, passa o valor 1500 para a função *exibe_titulo*:

```
#include <iostream.h>

struct Msg
 {
   char primeiro[256];
   void exibe_titulo(int valor)
   {
      cout << primeiro << valor << " Dicas sobre C/C++";
   }
 };

void main(void)
 {
   struct Msg livro = { "Este livro contém " };

   livro.exibe_titulo(1500);
 }
```

MÚLTIPLAS VARIÁVEIS DA MESMA ESTRUTURA 878

Na Dica 875, você definiu uma estrutura que continha uma função como um membro. O programa *func_mbr.cpp* então declarou uma variável do tipo estrutura. O programa a seguir, *multestr.cpp*, declara diversas variáveis do tipo *Msg*, atribui a cada uma delas uma string de caractere exclusiva, e, depois, exibe a mensagem usando a função *exibe_mensagem*:

```
#include <iostream.h>

struct Msg
 {
   char mensagem[256];
   void exibe_mensagem(void) { cout << mensagem; }
 };

void main(void)
 {
   struct Msg livro = { "Bíblia do Programador C/C++ " };
   struct Msg secao = { "Introdução à Linguagem C++ "  };

   livro.exibe_mensagem();
   secao.exibe_mensagem();
 }
```

DIFERENTES ESTRUTURAS COM OS MESMOS NOMES DE FUNÇÃO-MEMBRO 879

Como você aprendeu, C++ permite que seus programas coloquem funções como membros dentro de uma estrutura. Quando seus programas usam diferentes estruturas, algumas vezes duas estruturas têm os mesmos nomes de membro. O programa a seguir, *mesmonom.cpp*, cria duas estruturas diferentes, *Msg* e *MsgMaius*. Ambas as estruturas usam a função-membro *exibe_mensagem*. O compilador C++ diferencia entre os nomes de função das duas estruturas do mesmo modo que faz com membros de estruturas que não são funções, como mostrado aqui:

```
#include <iostream.h>
#include <string.h>

struct Msg
 {
   char mensagem[256];
   void exibe_mensagem(void) { cout << mensagem; }
 };

struct MsgMaius
 {
   char mensagem[256];
   void exibe_mensagem(void) { cout << strupr(mensagem); }
 };

void main(void)
 {
   Msg livro = { "Bíblia do Programador C/C++\n" };
   MsgMaius livro_mais = { "BÍBLIA DO PROGRAMADOR C/C++\n" };

   livro.exibe_mensagem();
   livro_mais.exibe_mensagem();
 }
```

As estruturas no programa *mesmonom.cpp* definem as funções dentro da própria estrutura. No entanto, na dica a seguir, você aprenderá como diferenciar entre as funções que seus programas definem fora das estruturas.

880 DIFERENTES FUNÇÕES COM OS MESMOS NOMES DE MEMBRO

Na dica anterior, você aprendeu que o compilador C++ distingue entre os membros de funções de diferentes tipos de estrutura. O programa a seguir, *nomedif.cpp*, define funções-membro fora de suas estruturas correspondentes. Para diferenciar entre as funções-membro, o programa precede cada definição de função com o nome da estrutura apropriada, seguido por dois-pontos dobrados:

```
#include <iostream.h>
#include <string.h>

struct Msg
 {
   char mensagem[256];
   void exibe_mensagem(void);
 };

struct MsgMaius
 {
   char mensagem[256];
   void exibe_mensagem(void);
 };

void Msg::exibe_mensagem(void)
 {
   cout << mensagem;
 }

void MsgMaius::exibe_mensagem(void)
 {
   cout << strupr(mensagem);
 }
```

```
void main(void)
 {
  Msg livro = {"Bíblia do Programador C/C++, do Jamsa!\n"};
  MsgMaius livro_mai = {"BÍBLIA DO PROGRAMADOR C/C++\n" };

  livro.exibe_mensagem();
  livro_mai.exibe_mensagem();
 }
```

COMPREENDENDO OS OBJETOS 881

No sentido mais simples, um *objeto* é uma coisa ou uma entidade do mundo real. Quando os programadores criam programas, eles escrevem instruções que trabalham com coisas diferentes, tais como variáveis ou arquivos. Diferentes objetos têm diferentes *operações* que seus programas efetuam nos objetos. Por exemplo, dado um objeto *arquivo*, seu programa poderia efetuar operações tais como ler, gravar ou imprimir o arquivo. Como você aprenderá, os programas C++ definem objetos em termos de uma classe. Uma *classe de objeto* (ou simplesmente uma "classe") define os dados que o objeto armazenará e as funções que operarão nos dados. Os programas C++ normalmente referenciarão as funções que manipulam os dados da classe como *métodos*. Por exemplo, a maioria dos seus programas C++ já usou os objetos *cin* e *cout*. No caso de *cin* e *cout*, o canal de E/S foi o objeto, e funções tais como *cin.get* e *cout.put* foram as operações no objeto.

COMPREENDENDO A PROGRAMAÇÃO ORIENTADA A OBJETOS 882

Para os programadores, um *objeto* é uma coleção de dados e um conjunto de operações, chamado *métodos*, que manipulam os dados. A *programação orientada a objetos* é o modo de ver os programas em termos de objetos (coisas) que formam um sistema. Após você ter identificado os objetos, pode determinar as operações que o sistema comumente efetua no objeto. Se você tem um objeto *documento*, por exemplo, as operações comuns poderiam incluir impressão, verificação ortográfica, fax ou até a eliminação. A programação orientada a objetos não requer uma linguagem de programação especial, tal como C++. Você pode escrever programas orientados a objetos em linguagens tais como COBOL ou FORTRAN. No entanto, como você aprenderá, as linguagens que os programadores descrevem como "orientadas a objetos" normalmente fornecem estruturas de dados de classe que permitem que seus programas agrupem dados e métodos em uma variável.

Como você aprenderá, a programação orientada a objetos tem muitas vantagens, duas das quais são a reutilização de objeto e a facilidade de compreensão. À medida que você for escrevendo mais programas, verá que pode normalmente usar os objetos que escreve em mais de um programa. Em vez de criar uma coleção de bibliotecas de funções, os programadores orientados a objetos criam *bibliotecas de classe*. Da mesma forma, quando você criar seus programas em torno de grupos de objetos e seus dados e métodos, você (e outros que lerem seus programas) compreenderá mais prontamente os programas orientados a objetos que os programas não-baseados em objetos (pelo menos após aprender a sintaxe da linguagem de programação que você usar). Os programadores e os profissionais da informática normalmente chamam C++ de uma extensão orientada a objetos da linguagem C. Muitas dicas a seguir examinam as capacidades baseadas em objetos de C++.

COMPREENDENDO POR QUE VOCÊ DEVE USAR OBJETOS 883

À medida que você começar a trabalhar com C++, precisará compreender por que é importante basear sua programação nos objetos. Embora os engenheiros de software estejam longe de concordar sobre o melhor uso dos objetos, a maioria deles concorda que o uso de objetos oferece as seguintes vantagens:

- **Facilidade de projeto e reutilização do código** — Após o código trabalhar corretamente, o uso de objetos aumenta sua capacidade de reutilizar um projeto ou código que criou para um aplicativo dentro de um segundo aplicativo.
- **Maior legibilidade** — Após você ter testado as bibliotecas de objetos, seu uso do código existente (funcional) aumentará a legibilidade do seu programa.
- **Facilidade de compreensão** — O uso de objetos ajuda os programadores a focalizar e a compreender os componentes fundamentais do sistema. O uso de objeto permite que os projetistas e programadores enfoquem as partes menores de um sistema e forneçam uma estrutura dentro da qual os projetistas

possam enfocar mais as operações que os programas efetuam nos objetos, as informações que os objetos precisam armazenar e outros componentes-chave do sistema.

- **Maior abstração** - A abstração permite que os projetistas e programadores "vejam o quadro geral" — ignorando temporariamente os detalhes subjacentes para que eles possam trabalhar com os elementos do sistema que são mais fáceis de compreender. Por exemplo, enfocando apenas os objetos do processador de texto discutidos na próxima dica, a implementação de um processador de texto pode se tornar muito menos intimidante.
- **Maior encapsulamento** - O *encapsulamento* (discutido na Dica 887, mais à frente) agrupa todas as partes de um objeto em um belo pacote. Por exemplo, a classe Livro definida anteriormente combina as funções e campos de dados que um programa precisa ter para trabalhar com um livro. Os programadores que estão trabalhando com a classe Livro não precisam conhecer cada parte da classe, somente aquela que eles precisam usar dentro de seus programas. A classe, por sua vez, trará com ela todas as partes necessárias.
- **Maior ocultação das informações** - A *ocultação das informações* é a capacidade de seu programa tratar uma função, procedimento ou um objeto como uma "caixa-preta", usando o item para efetuar uma operação específica sem ter que saber o que vai dentro. Por exemplo, no Capítulo 1, seus programas usaram os objetos *stream* para entrada e saída sem ter que compreender como os canais funcionam.

À medida que você examinar diferentes conceitos de programação C++ neste livro, aprenderá como os conceitos se relacionam com essas definições.

884 QUEBRANDO OS PROGRAMAS EM OBJETOS

No sentido mais simples, um objeto é uma coisa. Cachorros, livros e computadores são todos objetos. No passado, os programadores viam os programas como listas longas de instruções que efetuavam uma tarefa específica. Quando você cria programas orientados a objetos, olha para os objetos que constituem seu programa. Por exemplo, assuma que você esteja escrevendo um programa que implementa um processador de texto simples. Se você pensar em todas as operações que um processador de texto efetua, poderá rapidamente ficar desanimado. No entanto, se você vir o processador de texto como uma coleção de objetos distintos, o programa se tornará menos intimidante. Por exemplo, a Figura 884.1 ilustra os principais objetos no sistema do processador de texto.

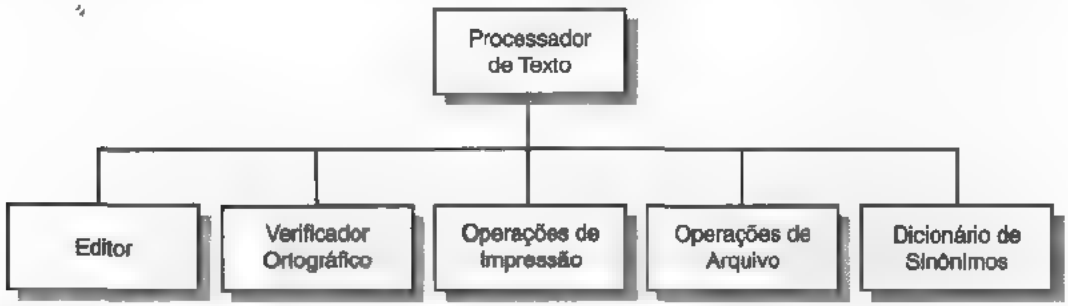

Figura 884.1 Vendo um processador de texto como uma coleção de objetos.

À medida que você examinar cada novo objeto, poderá descobrir que eles também são constituídos de outros objetos, como mostrado na Figura 884.2.

À medida que você começar a identificar os objetos que seu programa usa, verá que muitas partes diferentes do seu programa usam os mesmos tipos de objetos. Como resultado, quando você escrever seus programas em termos de objetos, poderá facilmente (e rapidamente) *reutilizar* o código que você escreve para uma seção em uma seção diferente do seu programa ou talvez até em um programa diferente. A reutilização do código é um dos recursos mais poderosos de C++.

Após você identificar os objetos, precisará determinar o propósito de cada objeto. Para fazer isso, pense nas operações que um objeto efetua ou nas operações que o programa efetua no objeto. Por exemplo, dado um objeto arquivo, um programa pode copiar, excluir ou renomear o arquivo. É importante notar que, em geral, essas operações se aplicam a cada arquivo no seu disco, independente do conteúdo do arquivo. Essas operações irão se tornar as *funções-membro* do objeto, para as quais você mais tarde escreverá funções C++ dentro de seu programa. Em seguida, identifique as informações que você precisa saber sobre o objeto. No caso do objeto arquivo,

você precisa conhecer o nome, tamanho, atributo de proteção e, possivelmente, a data e a hora em que o arquivo foi criado ou alterado pela última vez. Esses itens de dados irão se tornar as *variáveis-membro* do objeto arquivo. Conceitualmente, você pode agora visualizar seu objeto arquivo, como mostrado na Figura 884.3.

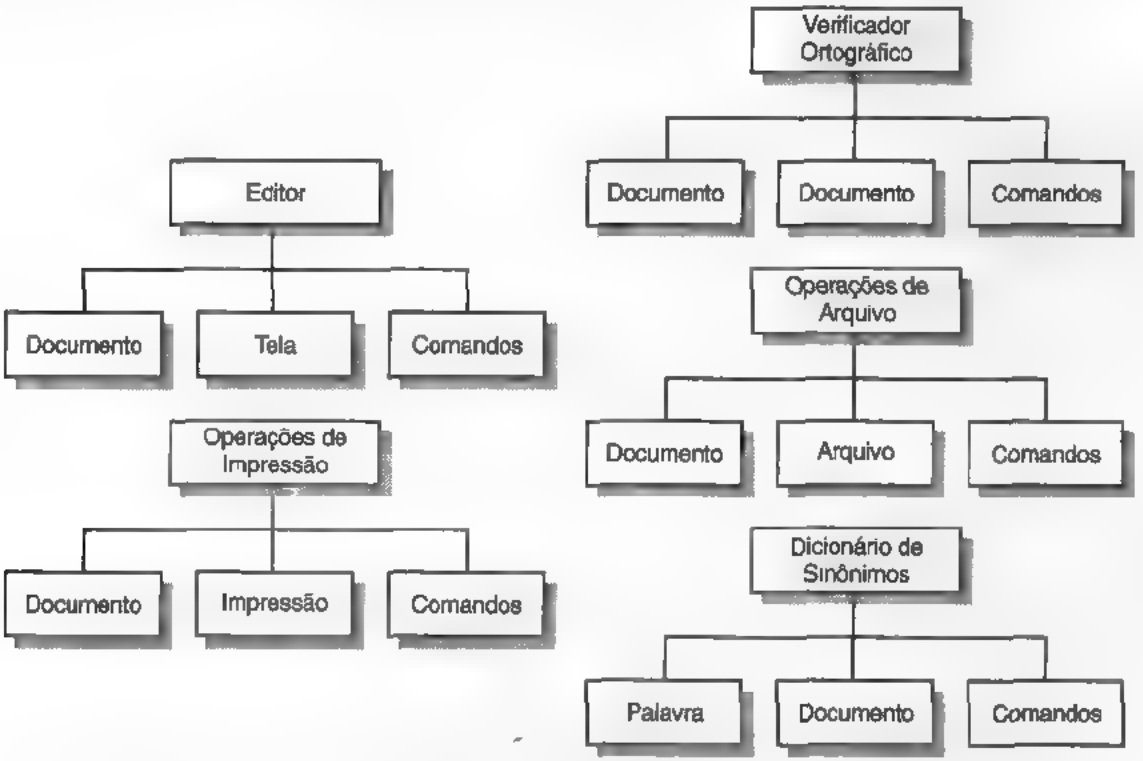

Figura 884.2 Identificando objetos adicionais dentro de seu processador de texto.

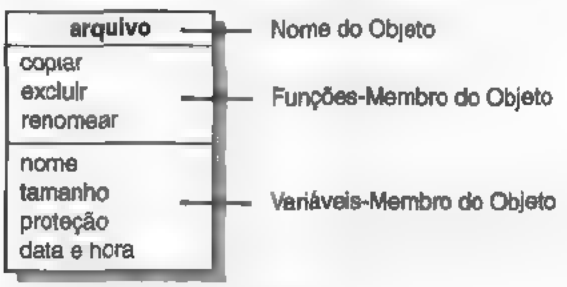

Figura 884.3 Funções-membro e variáveis de um objeto arquivo.

COMPREENDENDO OBJETOS E CLASSES 885

À medida que você for lendo artigos e livros sobre C++ e programação orientada a objetos, encontrará os termos *classe* e *objeto*. Uma *classe* fornece um gabarito, que define as funções-membro e os dados-membro que o tipo da classe requer. Um *objeto*, por outro lado, é uma *ocorrência*, ou exemplo específico de uma classe — basicamente uma variável *objeto*. Você precisa definir a classe antes da declaração do objeto.

Para declarar uma variável objeto, você simplesmente especifica o tipo da classe, seguido pelo nome da variável do objeto, como mostrado aqui:

```
nome_classe   nome_objeto;
```

Os programadores normalmente referenciam o processo de criar um objeto como *gerar uma ocorrência de um objeto*.

886 COMPREENDENDO AS CLASSES DE C++

Em todo este livro, os programas usaram estruturas para agrupar dados relacionados. Como você aprendeu, C++ permite que seus programas usem funções como membros da estrutura. Você pode visualizar melhor uma classe de C++ como uma extensão da estrutura. Uma classe, como uma estrutura, descreve um gabarito para declarações de variáveis futuras — ela não aloca memória para uma variável. Uma classe tem um nome e campos-membro. A definição a seguir, por exemplo, ilustra uma classe simples chamada *Livro*

```
class Livro
 {
   public:
     char titulo[256];
     char autor[64];
     float preco;
     void exibe_titulo(void)  { cout << titulo << '\n'; };
     float pega_preco(void)     { return(preco); };
 };
```

Como você pode ver, a definição de classe é muito similar a uma estrutura. O único item novo é o rótulo *public*. A Dica 896 discutirá o propósito do rótulo *public*. O programa a seguir, *primclass.cpp*, usa a classe *Livro* para exibir informações sobre um livro:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void)  { cout << titulo << '\n'; };
    float pega_preco(void)   { return(preco); };
 };

void main(void)
 {
   Livro dicas;

   strcpy(dicas.titulo, "Bíblia do Programador C/C++");
   strcpy(dicas.autor, "Jamsa e Klander");
   dicas.preco = 49.95;
   dicas.exibe_titulo();
   cout << "O preço do livro é " << setprecision(2) <<  dicas.pega_preco();
 }
```

887 COMPREENDENDO O ENCAPSULAMENTO

Lendo artigos e livros sobre programação orientada a objetos e C++, você poderá encontrar o termo *encapsulamento*. No sentido mais simples, encapsulamento é a combinação de dados e métodos em uma única estrutura de dados. O encapsulamento agrupa todos os componentes de um objeto. No sentido "orientado a objetos", o encapsulamento também define como ambos, o objeto e o resto do programa podem referenciar os dados de um objeto. Como você aprendeu, as classes de C++ lhe permitem dividir seus dados em seções public e private. Os programas somente podem acessar os dados privados de um objeto usando métodos públicos definidos. Agrupar juntos os dados de um objeto e dividir seus dados em seções públicas e privadas protege os dados de mau uso dos programas. Em C++, a classe é a ferramenta fundamental para o encapsulamento.

COMPREENDENDO O POLIMORFISMO 888

Em livros e artigos sobre C++, você frequentemente encontrará o termo *polimorfismo*. Polimorfismo permite que os programas apliquem a mesma operação para os objetos de diferentes tipos. Como o polimorfismo permite que os programadores apliquem a mesma operação a múltiplos tipos, ele permite que os programadores usem a mesma interface para acessar diferentes objetos. Em C++, as *funções virtuais* fornecem acesso ao polimorfismo. No sentido mais simples, a função virtual é um ponteiro para uma função que o compilador resolve em tempo de execução. Dependendo da função para a qual uma função virtual aponta, a operação que o programa efetua diferirá. Como resultado, uma única interface (a função virtual) poderá fornecer acesso a diferentes operações. A Dica 1090, mais à frente, discutirá as funções virtuais em detalhes.

COMPREENDENDO A HERANÇA 889

À medida que você derivar as classes usando o suporte de herança de C++, desenhar as figuras poderia lhe ajudar a compreender os relacionamentos entre as classes. Você verá que uma classe que deriva a partir de uma ou mais classes-base poderia tornar-se a classe-base para outras classes. À medida que você começar a definir suas classes, inicie com as características gerais e trabalhe voltado para os detalhes específicos ao derivar as novas classes. Por exemplo, se você estivesse derivando classes para tipos de cachorros, sua primeira classe-base poderia simplesmente ser *Cachorros*. *Cachorros* conteria características comuns a todas as raças de cães, tais como nome, origem, altura, peso e cor. Seu próximo nível poderia se tornar mais refinado ao criar as classes. Os tipos de classe de segundo nível, *CachorrosComManchas* e *CachorrosSemManchas*, por exemplo, herdariam as características comuns que você definiu na classe-base *Cachorros*. No entanto, à medida que você refinar mais os pedigrees (por exemplo, entre Dálmatas e Labradores), poderá usar essas classes de segundo nível como classes-base para definições de outras classes. Seus níveis de classe-base crescerão, conceitualmente similar ao crescimento da árvore de uma família, como mostrado na Figura 889.

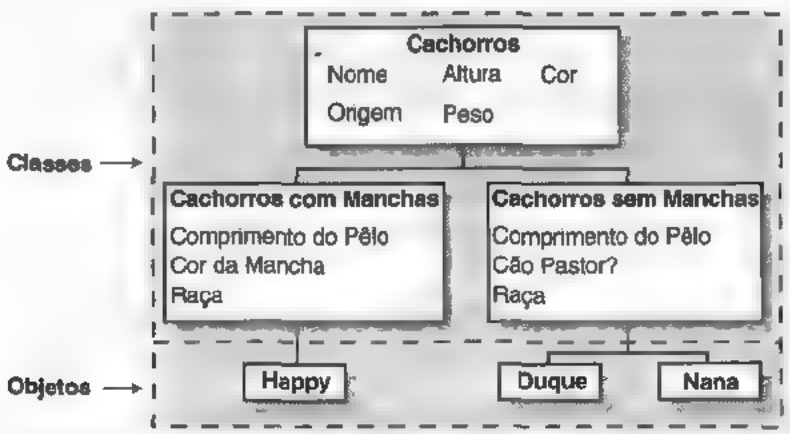

Figura 889 A árvore de herança de Cachorros.

DECIDINDO ENTRE CLASSES E ESTRUTURAS 890

Anteriormente, neste livro, já foi visto as estruturas de C. Como você está estudando classes, deve reconhecer que a sintaxe para trabalhar com classes é muito similar à sintaxe que você usa com as estruturas de C. Você pode estar imaginando quando deverá usar classes, em vez de estruturas, ou até mesmo uniões. Como sabe, as classes, estruturas e uniões permitem que seus programas armazenem dados relacionados. Seus programas deverão usar classes sempre que efetuarem operações específicas nos dados. Por exemplo, considere que, caso simplesmente necessite armazenar uma data, é possível usar uma estrutura ou uma união. No entanto, se você quiser que o programa formate e exiba a data, arquive a data ou compare duas datas, você deverá usar uma classe. Da mesma forma, se você precisa escolher entre usar uma estrutura ou uma união, deve basear sua decisão no número de valores que a estrutura de dados precisa armazenar em qualquer dado momento. Finalmente, lembre-se de que, por padrão, os membros de classe são privados, e os membros de estruturas e uniões são públicos.

Se você experimentar as estruturas de C++, verá que elas suportam muitos dos mesmos recursos que as classes de C++, *tais como dados públicos e privados, funções-membro e assim por diante. Como regra, se você estiver criando objetos, use uma classe.*

891 CRIANDO UM MODELO SIMPLES DE CLASSE

O melhor modo de compreender as classes e os objetos de C++ é criar um programa simples. Na próxima dica, o programa de exemplo, *filmes.cpp*, cria uma classe chamada *filme*. Depois ele cria dois objetos do tipo *filme*, chamados *fugitivo* e *insone*. O programa define a classe *filme*, como mostrado aqui:

```
class filme
 {
   public:
     char nome[64];
     char primeiro_astro[64];
     char segundo_astro[64];
     void exibe_filme(void);
     void inicializa(char *nome, char *primeiro, char *segundo);
 };
```

Como você pode ver, a classe *filme* usa três variáveis-membro e duas funções-membro. Seguindo a definição de classe, o programa precisa definir as funções-membro *exibe_filme* e *inicializa*, como mostrado aqui:

```
void filme::exibe_filme(void)
 {
   cout << "Nome do filme: " << nome << endl;
   cout << "Estrelando: " << primeiro_astro << " e " << segundo_astro
 << endl << endl;
 }

void filme::inicializa(char *filme_nome, char *primeiro, char *segundo)
 {
   strcpy(nome, filme_nome);
   strcpy(primeiro_astro, primeiro);
   strcpy(segundo_astro, segundo);
 }
```

As definições das funções da classe são muito similares às definições de funções padrão. No entanto, existem duas diferenças principais. Primeiro, o nome da classe e dois-pontos dobrados precedem os nomes das funções:

```
void filme::inicializa(char *nome_filme, char *primeiro, char *segundo)
```

Segundo, dentro da função da classe, os comandos podem referenciar diretamente as variáveis-membro da classe, como a seguir:

```
void filme::inicializa(char *nome_filme, char *primeiro, char *segundo)
 {
   strcpy(nome, nome_filme);
   strcpy(primeiro_astro, primeiro);
   strcpy(segundo_astro, segundo);
 }
```

892 IMPLEMENTANDO O PROGRAMA DA CLASSE SIMPLES

Na dica anterior, você aprendeu sobre os componentes da classe simples *filme*. Agora que você compreende como criar uma classe, precisa implementar a classe para compreender melhor como trabalhará com a classe dentro de seus programas. O programa *filmes.cpp* implementa a classe *filme*, como mostrado aqui.

```
#include <iostream.h>
#include <string.h>
```

```
class filme
 {
   public:
     char nome[64];
     char primeiro_astro[64];
     char segundo_astro[64];
     void exibe_filme(void);
     void inicializa(char *nome, char *primeiro, char *segundo);
 };

void filme::exibe_filme(void)
 {
   cout << "Nome do filme: " << nome << endl;
   cout << "Estrelando: " << primeiro_astro << " e " << segundo_astro
<< endl << endl;
 }

void filme::inicializa(char *filme_nome, char *primeiro, char *segundo)
 {
   strcpy(nome, filme_nome);
   strcpy(primeiro_astro, primeiro);
   strcpy(segundo_astro, segundo);
 }

void main(void)
 {
   filme fugitivo, insone;

   fugitivo.inicializa("O Fugitivo", "Harrison Ford", "Tommy Lee Jones");
   insone.inicializa("Insone em Seattle", "Tom Hanks", "Meg Ryan");
   fugitivo.exibe_filme();
   insone.exibe_filme();
 }
```

Como você pode ver, o programa cria dois objetos do tipo *filme:*

```
filme fugitivo, insone;
```

No programa *filmes cpp*, o programa usa a função-membro *inicializa* para inicializar as variáveis-membro da classe Em dicas posteriores você aprenderá como usar as funções construtoras para inicializar as variáveis-membro de um modo mais natural.

DEFININDO OS COMPONENTES DE UMA CLASSE 893

Como você aprendeu, uma classe consiste de um ou mais componentes distintos, que podem ser variáveis, funções, ou ambas. Uma *declaração de classe* define um novo tipo de classe que vincula código e dados. Seus programas usarão o novo tipo para declarar objetos dessa classe. Portanto, uma classe é uma abstração lógica, mas um objeto tem existência física. Em outras palavras, um objeto é uma *ocorrência* de uma classe.

Como você viu nas dicas anteriores, uma declaração de classe é similar em sintaxe à declaração de uma estrutura. O código a seguir mostra a forma geral de uma classe:

```
class nome_classe
 {
   funções e dados privados
especificador_acesso:
   funções e dados privados
especificador_acesso:
   funções e dados privados
.
.
.
```

```
especificador_acesso:
   funções e dados privados
} lista_objetos;
```

A *lista_objetos* é opcional. Se a *lista_objeto* estiver presente, ela declarará objetos da classe. O *especificador_acesso* é uma das três palavras-chave de definição de classe de C++ que você aprendeu anteriormente, *public*, *private* e *protected*.

Os dados e funções públicas que constituem a classe são comumente referenciados como *propriedades* e *métodos*. Como você aprendeu, os métodos também são conhecidos como *funções de interface*.

894 COMPREENDENDO O OPERADOR DE DEFINIÇÃO DE ESCOPO

Você já sabe que seus programas usarão o *operador de definição de escopo* (o operador ::) para vincular um nome de classe com um nome de membro para dizer ao compilador a qual classe o membro pertence. O operador de definição de escopo também permite que seus programas acessem um nome em um escopo fechado que uma declaração local do mesmo nome oculta. Por exemplo, considere o seguinte fragmento de código.

```
// Comandos do programa
//
int i;           // global
void f();
   int i;        // local
   i = 10;       // refere-se à variável i local
//
// Comandos do programa
```

No entanto, se a função *f* requisesse uma referência à variável global *i*, em vez da local *i*, você poderia reescrever o fragmento, como mostrado aqui:

```
// Comandos do programa
//
int i;           // global
void f();
   int i;        // local
   ::i = 10;     // refere-se à variável i global não a local i
//
// Comandos do programa
```

895 USANDO OU OMITINDO O NOME DA CLASSE DAS DECLARAÇÕES

Como você aprendeu, uma classe define um gabarito para futuras declarações de variáveis. Após você definir uma classe, seu programa pode declarar uma classe em uma dentre duas formas — seu programa pode usar a classe, ou o programa pode simplesmente especificar o nome da classe, como a seguir:

```
class Livro
 {
   public:
     char titulo[256];
     char autor[64];
     float preco;
     void exibe_titulo(void) { cout << titulo << '\n'; };
     float le_preco(void)    { return(preco); };
 };

// Declara variáveis do tipo da classe
class Livro dicas;
Livro diario;
```

Como você pode ver, C++ usa os nomes de classes da mesma maneira que usa os nomes de estruturas: para criar um tipo com o qual você pode mais tarde declarar outras variáveis.

COMPREENDENDO O RÓTULO PUBLIC: 896

Na dica anterior você criou uma classe simples, chamada *Livro*, que contém o rótulo *public:*, como mostrado aqui:

```
class Livro
 {
   public:
     char titulo[256];
     char autor[64];
     float preco;
     void exibe_titulo(void) { cout << titulo << '\n'; };
     float le_preco(void)    { return(preco); };
 };
```

Ao contrário de uma estrutura, cujos membros são todos acessíveis a um programa, uma classe pode ter membros que o programa pode acessar diretamente usando o operador *ponto*, e outros membros (chamados *membros privados*) que o programa não pode acessar diretamente. O rótulo *public:* identifica os membros de classe que o programa pode acessar usando o operador ponto. Se você quiser que o programa acesse um membro específico diretamente, precisará declarar o membro dentro dos membros públicos da classe.

COMPREENDENDO A OCULTAÇÃO DE INFORMAÇÕES 897

Ocultação de informações é o processo de ocultar detalhes subjacentes da implementação de uma função, programa ou classe. A ocultação de informações permite que os programadores tratem as funções e as classes como *caixas-pretas*. Em outras palavras, se um programador passa um valor para uma função, o programa sabe que um resultado específico ocorrerá. O programador não precisa saber como a função calcula o resultado, mas simplesmente que a função funciona. Por exemplo, a maioria dos programadores não conhece a matemática efetuada pela função *tanh*, que retorna a tangente hiperbólica de um ângulo. No entanto, os programadores sabem que se passarem um valor específico para a função, um resultado conhecido ocorrerá. Para usar a função, o programador somente precisará conhecer os parâmetros de entrada e os valores que a função retornará.

Na programação orientada a objetos, um objeto pode ter detalhes subjacentes de implementação. Por exemplo, o Microsoft *Word*®, *ou o Excel*® ou algum outro programa pode armazenar dados em um *documento*. No entanto, para usar o objeto *documento*, o programa não deve precisar conhecer o formato. Em vez disso, o programa deve efetuar operações de leitura, gravação, impressão e transmissão via fax sem conhecer os detalhes do objeto. Para ajudar os programadores a ocultar os detalhes subjacentes de objetos, C++ lhe permite dividir uma definição de classe em partes privada e pública. O programa pode acessar diretamente dados e métodos públicos, mas não pode acessar diretamente os dados e métodos privados

COMPREENDENDO O RÓTULO PRIVATE: 898

Você já viu que C++ lhe permite dividir uma definição de classe em seções públicas e privadas. O programa pode usar o operador *ponto* (.) para acessar dados e métodos públicos. No entanto, o programa não pode usar o operador ponto para acessar diretamente os dados e métodos privados. A definição de classe a seguir expande a classe *Livro* para incluir dados e métodos públicos e privados:

```
class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void)  { cout << titulo << '\n'; };
    float pega_preco(void)   { return(preco); };
    void exibe_livro(void)
    {
```

```
        exibe_titulo();
        exibe_editora();
    };
    void atribui_editora(char *nome) { strcpy(editora, nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n'; };
};
```

O programa pode usar o operador ponto para acessar diretamente os métodos e dados que residem na seção pública. No entanto, o único modo de acessar os dados e métodos privados é por meio dos métodos públicos. A Dica 900, à frente, apresenta um programa que manipula os dados privados e públicos.

899 USANDO O RÓTULO PROTECTED:

Como você aprendeu, C++ lhe permite categorizar os membros de classe como públicos ou privados. A *categorização de privado ou público de um membro controla como seu programa pode acessar o membro.* Quando você usa herança para derivar uma classe de outra, C++ acrescenta uma terceira categoria de membro: *protected*. Um *membro protected* está essencialmente no meio de um membro privado e público. Para uma classe-base, os objetos derivados podem acessar os membros protegidos exatamente como se os membros fossem públicos. No entanto, fora dos objetos derivados, somente as rotinas de interface pública podem acessar os membros protegidos. O código a seguir acrescenta dois membros protegidos à definição da classe *Livro:*

```
class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo, titulo); };
    void exibe_titulo(void)  { cout << titulo << endl; };
  protectd:
    float custo;
    void exibe custo(void)  { cout << custo << endl);
  private:
    char titulo[64];
 };
```

No exemplo anterior, os objetos derivados da classe *Livro* podem acessar os membros *custo* e *exibe_custo* exatamente como se os membros fossem públicos. No entanto, fora das classes derivadas, o programa precisa tratar os membros como se eles fossem privados.

900 USANDO DADOS PÚBLICOS E PRIVADOS

C++ lhe permite dividir uma definição de classe em dados e métodos *públicos* e *privados*. Os programas podem acessar os dados e métodos públicos usando o operador *ponto*. No entanto, para acessar os dados e métodos privados, o programa precisa chamar os métodos públicos. O programa não pode manipular ou chamar diretamente os dados e métodos privados. O programa a seguir, *pub_priv.cpp*, ilustra o uso de dados públicos e privados:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void) { cout << titulo << '\n'; };
    float pega_preco(void)   { return(preco); };
    void exibe_livro(void)
    {
```

```
        exibe_titulo();
        exibe_editora();
      };
      void atribui_editora(char *nome) { strcpy(editora, nome); };
    private:
      char editora[256];
      void exibe_editora(void) { cout << editora << '\n'; };
  };

void main(void)
  {
    Livro biblia;

    strcpy(biblia.titulo, "Bíblia do Programador C/C++");
    strcpy(biblia.autor, "Jamsa e Klander");
    biblia.preco = 49.95;
    biblia.atribui_editora("Makron Books");
    biblia.exibe_livro();
  }
```

Como você pode ver, o método público *atribui_editora* inicializa o membro privado *editora*. Se o programa tivesse tentado acessar o membro *editora* diretamente, o compilador teria gerado um erro. De um modo similar, o programa usa o método público *exibe_livro*, o qual, por sua vez, chama o método privado *exibe_editora*. Novamente, o programa não pode acessar diretamente o método privado.

DETERMINANDO O QUE OCULTAR E O QUE TORNAR PÚBLICO 901

Você sabe que C++ lhe permite dividir as definições de classes em seções privadas e públicas. Uma das tarefas mais difíceis que os programadores novos na programação orientada a objetos enfrentam é determinar quais membros de cada classe eles devem ocultar e quais membros devem tornar públicos. Como uma regra geral, quanto menos um programa souber sobre uma classe, melhor. Portanto, você deve tentar usar dados e métodos privados tão freqüentemente quanto possível. Quando você usa dados e métodos privados, os programas que usam o objeto têm que usar os métodos públicos do objeto para acessar os dados do objeto. Como você aprenderá na dica a seguir, forçar os programas a manipular os dados do objeto usando somente métodos públicos pode reduzir os erros de programação. Em outras palavras, você normalmente não quer que um programa manipule os dados de um objeto diretamente usando apenas o operador ponto. Fazer uso de dados privados desse modo melhora a ocultação das informações.

OS MÉTODOS PÚBLICOS SÃO FREQÜENTEMENTE CHAMADOS DE FUNÇÕES DE INTERFACE 902

Como você aprendeu na dica anterior, seus programas devem tentar colocar a maioria dos dados de um objeto na seção privada de uma definição de classe. Quando seus programas colocam dados de objeto em seções privadas, outros programas podem acessar os dados somente chamando os métodos da classe pública. Desse modo, os métodos públicos fornecem a *interface* do seu programa para os dados do objeto. Usando essas funções de interface, seus programas podem verificar que o valor que o programa quer atribuir a um membro é válido. Por exemplo, assuma que o membro *derretendo* na classe *ReatorNuclear* somente deva conter os valores de 1 até 5. Se o membro for público, um programa poderia atribuir um valor inválido usando o operador ponto, como a seguir:

```
nuclear.derretendo = 99;
```

Restringindo o acesso ao membro *derretendo* para o método público *def_derretendo*, o objeto pode verificar o valor, como mostrado a seguir:

```
int def_derretendo(int valor)
  {
    if ((valor >= 1) && (valor <= 5))
```

```
    {
      nuclear.derretendo = valor;
      return(0);
    }
  else
     return(-1);     // Valor inválido
}
```

Restringindo o acesso aos dados do objeto para os métodos públicos, as únicas operações que um programa pode efetuar nos dados dentro do objeto são aquelas que o próprio objeto define.

903 DEFININDO FUNÇÕES DE CLASSE FORA DA CLASSE

Várias dicas precedentes criaram classes simples que definiram funções-membro dentro das próprias classes. Aumentando o tamanho das suas funções de classe, você eventualmente definirá as funções fora da classe. O programa a seguir, *livrofun.cpp*, define as funções para o objeto *Livro* fora da classe. Como você verá, o programa identifica as funções de classe precedendo cada nome de função com o nome da classe e dois-pontos dobrados, como mostrado aqui:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void);
    float pega_preco(void);
    void exibe_livro(void);
    void atribui_editora(char *nome);
  private:
    char editora[256];
    void exibe_editora(void);
 };

void Livro::exibe_titulo(void)
 { cout << titulo << '\n'; };

float Livro::pega_preco(void)
 { return(preco); };

void Livro::exibe_Livro(void)
 {
   exibe_titulo();
   exibe_editora();
 };

void Livro::atribui_editora(char *nome)
  { strcpy(editora, nome); };

void Livro::exibe_editora(void)
  { cout << editora << '\n'; };

void main(void)
 {
   Livro dicas;
```

```
    strcpy(dicas.titulo, "Bíblia do Programador C/C++!");
    strcpy(dicas.autor, "Jamsa e Klander");
    dicas.preco = 49.95;
    dicas.atribui_editora("Makron Books");
    dicas.exibe_Livro();
}
```

DEFININDO MÉTODOS DENTRO E FORA DAS CLASSES 904

Como você aprendeu, C++ lhe permite definir métodos dentro e fora da declaração da classe. A decisão que você faz sobre onde definir a função do método afeta o código que o compilador cria para o programa. Quando você definir um método dentro de uma classe, o compilador tratará cada referência do método como uma chamada de função *inline*, colocando as instruções correspondentes da função no código-objeto em cada referência de método. Como você aprendeu, usar código inline pode melhorar o desempenho, mas, também, aumenta o tamanho do seu programa. Por outro lado, quando você define uma função fora da classe, o compilador não usa código inline. Em vez disso, o compilador gerará código para uma função que o programa chamará em cada referência de método. Portanto, se sua classe tiver uma operação comum que é pequena, você pode querer que o compilador gere código inline para o método. Se o método for maior, não instrua o compilador a gerar código inline.

COMPREENDENDO AS OCORRÊNCIAS DO OBJETO 905

Muitos livros e artigos sobre C++ referenciam as *ocorrências dos objetos*. Em resumo, uma ocorrência de objeto é uma variável objeto. Como você aprendeu, uma classe define um gabarito para declarações futuras de variáveis. Quando você mais tarde, declarar um objeto, criará uma ocorrência do objeto. Em outras palavras, quando o compilador aloca memória para uma variável, seu programa cria uma ocorrência do objeto. Todas as ocorrências da mesma classe têm as mesmas características. Para os propósitos deste livro, uma ocorrência é uma variável de uma classe específica.

AS OCORRÊNCIAS DE OBJETOS DEVEM COMPARTILHAR O CÓDIGO 906

C++ lhe permite definir métodos de classe dentro ou fora da classe. Quando você declara métodos de classe fora da classe, as ocorrências compartilham a mesma cópia dos métodos. Por exemplo, se você tiver uma classe com três métodos, e criar 100 ocorrências dessa classe, seus programas conterão somente os três métodos. No entanto, se você incluir código in-line, as ocorrências não compartilharão o código. Portanto, você deve reservar código in-line para operações pequenas e comumente executadas, em que o desempenho da operação é mais importante que o tamanho do programa. Por exemplo, o programa a seguir, *cplivro.cpp*, cria duas ocorrências da classe *Livro*:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
 public:
   char titulo[256];
   char autor[64];
   float preco;
   void exibe_titulo(void) { cout << titulo << '\n'; };
   float pega_preco(void) { return(preco); };
   void exibe_livro(void);
   void assign_editora(char *nome) { strcpy(editora, nome); };
 private:
   char editora[256];
   void exibe_editora(void) { cout << editora <<'\n'; };
};
```

```
void Livro::exibe_livro(void)
 {
    exibe_titulo();
    exibe_editora();
 };

void main(void)
 {
   Livro dicas, diario;

   strcpy(dicas.titulo, "Bíblia do Programador C/C++!");
   strcpy(dicas.autor, "Jamsa e Klander");
   dicas.preco = 49,95;
   dicas.assign_editora("Makron Books");

   strcpy(diario.titulo, "Todos os Meus Segredos...");
   strcpy(diario.autor, "Kris Jamsa");
   diario.preco = 9.95;
   diario.assign_editora("Nenhuma");

   dicas.exibe_livro();
   diario.exibe_livro();
 }
```

Se você compilar o programa *cprlivro.cpp* usando o *Turbo C++ Lite* para produzir uma listagem em linguagem assembly, verá que as ocorrências não compartilham o código para os métodos in-line, mas, sim, o código definido fora da classe.

907 ACESSANDO OS MEMBROS DA CLASSE

Nas dicas anteriores, você usou o operador ponto para chamar as funções-membro da classe. Quando seus programas colocam objetos-membro após um rótulo *public*, seus programas podem acessar os membros usando os operadores ponto. Por exemplo, na Dica 892 você criou uma classe *filme* simples e acessou sua função-membro *exibe_filme*. O programa a seguir, *publico.cpp*, usa a função *inicializa* para atribuir valores aos membros dos objetos *fugitivo* e insone. O programa então exibe os diferentes valores-membro referenciando o membro usando o operador ponto, como mostrado aqui:

```
#include <iostream.h>
#include <string.h>

class filme
 {
   public:
     char nome[64];
     char primeiro_astro[64];
     char segundo_astro[64];
     void exibe_filme(void);
     void inicializa(char *nome, char *primeiro, char *segundo);
 };

void filme::exibe_filme(void)
 {
   cout << "Nome do filme: " << nome << endl;
   cout << "Estrelando: " << primeiro_astro << " e "
        << segundo_astro << endl << endl;
 }

void filme::inicializa(char *filme_nome, char *primeiro, char *segundo)
 {
   strcpy(nome, filme_nome);
   strcpy(primeiro_astro, primeiro);
   strcpy(segundo_astro, segundo);
```

```
}

void main(void)
{
  filme fugitivo, insone;

  fugitivo.inicializa("O Fugitivo", "Harrison Ford", "Tommy Lee Jones");
  insone.inicializa("Insone em Seattle", "Tom Hanks", "Meg Ryan");
  cout << "Os dois últimos filmes a que assisti foram: "
       << fugitivo.nome << " e " << insone.nome << endl;
  cout << "Achei que o " << fugitivo.primeiro_astro
       << " estava ótimo!" << endl;
}
```

Como os membros da classe são *públicos*, os programas podem acessar os membros diretamente. Quando você compilar e executar o programa *publico.cpp*, sua tela exibirá o seguinte:

```
Os dois últimos filmes a que assisti foram: O Fugitivo e Insone em Seattle
Achei o Harrison Ford excelente no papel!
C:\>
```

Quando uma classe define variáveis-membro como públicas, seus programas podem acessar as variáveis-membro usando o operador ponto. No entanto, como você aprenderá em dicas posteriores, esse acesso direto ao membro da variável nem sempre é o preferível.

REVISANDO O OPERADOR DE DEFINIÇÃO GLOBAL 908

Como você aprendeu, C++ lhe permite repetir os nomes das funções e variáveis em múltiplas classes. Em outras palavras, a existência da função *prox_dia* na classe *semana* não impede sua existência dentro da classe *mes*. Dentro do seu programa, você pode usar o *operador de definição global* (::) para se proteger contra a confusão de nomes dos dados e das funções. Quando você precisar referenciar um membro de classe (dados ou função), simplesmente preceda o nome do membro com o nome da classe e os dois-pontos dobrados, como mostrado aqui:

```
void filme::inicializa(char *nome, char *primeiro_astro,
                       char *segundo_astro)
{
  strcpy(filme::nome, nome);
  strcpy(filme::primeiro_astro, primeiro_astro);
  strcpy(filme::segundo_astro, segundo_astro);
}
```

INICIALIZANDO OS VALORES DA CLASSE 909

Como você viu nas dicas anteriores, é importante inicializar os valores dentro de suas classes toda vez que você gera a ocorrência de um novo objeto a partir da classe. C++ fornece vários modos de inicializar os valores dentro de suas classes. Em dicas posteriores você aprenderá sobre as funções construtoras, que os programadores C++ comumente usam para inicializar novas ocorrências de classes. No entanto, você também pode criar uma função-membro para inicializar uma classe, como a seguir:

```
void filme::inicializa(char *nome, char *primeiro_astro,
                       char *segundo_astro)
{
  strcpy(filme::nome, nome);
  strcpy(filme::primeiro_astro, primeiro_astro);
  strcpy(filme::segundo_astro, segundo_astro);
}
```

910 USANDO OUTRO MÉTODO PARA INICIALIZAR OS VALORES DE CLASSE

Você pode inicializar os valores de classe dentro da inicialização de uma função. Como aprenderá em dicas posteriores, também é possível inicializar suas classes dentro de funções construtoras. No entanto, à medida que você examinar programas C++, poderá encontrar uma técnica de inicialização de membro único. Por exemplo, assuma que você queira que a construtora *contador* inicialize a variável *conta* para 0:

```
contador::contador(void)
 {
   conta = 0;
   // Outros comandos
 }
```

Em vez de lhe forçar a criar uma função de inicialização separada para cada classe, C++ lhe permite inicializar as variáveis-membro da classe colocando o nome da variável e o valor desejado após dois-pontos e antes dos comandos da função, como mostrado aqui:

```
contador::contador(void) : conta(0)
 {
   // Outros comandos
 }
```

O programa a seguir, *con_inic.cpp*, usa o formato de inicialização de construtora para inicializar três variáveis-membro com os valores 1, 2 e 3:

```
#include <iostream.h>

class objeto
 {
   public:
     objeto::objeto(void);
     void show_objeto(void);
   private:
     int a;
     int b;
     int c;
 };

objeto::objeto(void) : a(1), b(2), c(3) { };

void objeto::show_objeto(void)
 {
   cout << "a contém: " << a << endl;
   cout << "b contém: " << b << endl;
   cout << "c contém: " << c << endl;
 }

void main(void)
 {
   objeto numeros;
   numeros.show_objeto();
 }
```

Quando você compilar e executar o programa *con_inic.cpp*, sua tela exibirá a seguinte saída:

```
a contém 1
b contém 2
c contém 3
C:\>
```

COMPREENDENDO OS MEMBROS DE CLASSE ESTÁTICA 911

Dentro de suas classes de C++, você pode definir dados e funções-membro como estáticos. Como você aprenderá, declarar um dado ou uma função-membro como *static* tem importantes implicações para suas classes. C++ governa os dados-membro estáticos e as funções por regras diferentes do que com os membros e funções normais. Por exemplo, uma função estática somente pode acessar outros membros estáticos dentro da mesma classe (bem como funções globais e dados-membro). Antes de você usar membros estáticos dentro de suas classes, é importante compreender as implicações. As Dicas 912 e 913 discutem os dados-membro e as funções estáticas em detalhes.

USANDO DADOS-MEMBRO ESTÁTICOS 912

Na dica anterior você viu que o compilador C++ trata os dados-membro que você precede com a palavra-chave *static* diferentemente de como trata os dados-membro normais. Na verdade, quando você precede a declaração de uma variável-membro com a palavra-chave *static*, está dizendo ao compilador que somente uma cópia da variável existirá e que todos os objetos da classe compartilharão essa variável. Ao contrário dos dados-membro normais, o programa não cria cópias individuais de um membro estático para cada objeto. Independentemente de quantos objetos de uma classe o programa crie, somente uma cópia de cada variável-membro estática existirá. Portanto, todos os objetos dessa classe usam a mesma variável. O compilador inicializa todas as variáveis estáticas com zero quando o programa cria a primeira ocorrência de objeto.

Ao declarar um membro de dados estático dentro de uma classe, você não *define* esse membro. Em outras palavras, você não aloca armazenamento na memória para o membro. Em vez disso, você precisa fornecer uma definição global para o membro de dados estáticos em outra parte, fora da classe. Para fornecer a definição global, você irá redeclarar o membro de dados estáticos usando o operador de definição de escopo. Fazer isso instrui o compilador a alocar espaço para o membro estático. Para compreender melhor o uso e o efeito de um dado membro estático, considere o seguinte programa, *mem_esta.cpp:*

```
#include <iostream.h>

class compartilhada
 {
  static int a;
    int b;
  public:
    void set(int i, int j)
    {
      a=i;
      b=j;
    }
    void show();
 };

int compartilhada::a;   // Define a variável global a

void compartilhada::show()
 {
   cout << "Isto é estático a: " << a << endl;
   cout << "Isto é não-estático b: " << b << endl;
 }

void main(void)
 {
   compartilhada x, y;

   x.set(1,1);
   x.show();
   y.set(2,2);
   y.show();
```

```
    x.show();
}
```

Quando você compilar e executar o programa *mem_esta.cpp*, ele gerará o seguinte resultado:

```
Isto é a estática a:  1
Isto é a não-estática b: 1
Isto é a estática a:  2
Isto é a não-estática b: 2
Isto é a estática a:  2
Isto é a não-estática b: 1
```

913 USANDO FUNÇÕES-MEMBRO ESTÁTICAS

Você já sabe que é possível declarar dados-membro dentro de suas classes como estáticas. Também é possível declarar funções-membro dentro de suas classes como estáticas. O compilador C++ restringe as funções que você declara como estática de vários modos:

1. As funções estáticas podem acessar somente outros membros estáticos da classe.
2. As funções-membro estáticas não têm um ponteiro *this*.
3. Você não pode sobrecarregar uma função estática com uma função não-estática, ou vice-versa.

O programa a seguir, *fun_esta.cpp*, é uma versão retrabalhada do programa *mem_esta.cpp* que apareceu na dica anterior. O programa *fun_esta.cpp* declara *exibe* como estática, de modo que o programa pode acessar *exibe* somente usando o operador de definição de classe, ou em conexão com um único objeto, como mostrado aqui:

```
#include <iostream.h>

class compartilhada
 {
  static int a;
    int b;
  public:
    void define(int i, int j)
    {
      a=i;
      b=j;
    }
    static void exibe();
 };

int compartilhada::a;  // Define a variável global a

void compartilhada::exibe()
 {
   cout << "Isto é a estática a: " << a << endl;
 }

void main(void)
 {
   compartilhada x, y;

   x.define(1,1);
   y.define(2,2);
   compartilhada::exibe();
   y.exibe();
   x.exibe();
 }
```

COMPREENDENDO AS DECLARAÇÕES DAS FUNÇÕES-MEMBRO 914

Como você aprendeu, uma classe contém variáveis-membro e funções-membro Quando você define funções de classe, pode definir as funções fora da definição da classe, como mostrado aqui:

```
class filme
 {
   public:
     char nome[64];
     char primeiro_astro[64];
     char segundo_astro[64];
     void exibe_filme(void);
     void inicializa(char *nome, char *primeiro, char *segundo);
 };

void filme::exibe_filme(void)
 {
   cout << "Nome do filme: " << nome << endl;
   cout << "Estrelando: " << primeiro_astro << " e " << segundo_astro
   << endl << endl;
 }

void filme::inicializa(char *filme_nome, char *primeiro, char *segundo)
 {
   strcpy(nome, filme_nome);
   strcpy(primeiro_astro, primeiro);
   strcpy(segundo_astro, segundo);
 }
```

Neste caso, a definição da classe precisa conter protótipos que descrevem cada função-membro da classe. Da mesma forma, as definições da função precisam especificar o nome da classe antes do nome da função.

USANDO DECLARAÇÕES DE FUNÇÃO IN-LINE 915

Na dica anterior, você aprendeu como definir as funções de classe fora de uma definição de classe. Você também pode definir suas funções-membro de classe dentro da classe, na verdade colocando os comandos da função dentro da declaração da classe. Por exemplo, o programa a seguir, *inline.cpp*, define as funções-membro da classe inline, dentro da declaração da classe:

```
#include <iostream.h>
#include <string.h>

class filme
 {
public:
  char nome[64];
  char primeiro astro[64];
  char segundo astro[64];
  void exibe_filme(void)
    {
      cout << "Nome do filme: " << nome << endl;
      cout << "Estrelando: " << primeiro_astro << " e " << segundo_astro
      << endl << endl;
    }
  void inicializa(char *filme_nome, char *primeiro, char *segundo)
    {
      strcpy(nome, filme_nome);
      strcpy(primeiro_astro, primeiro);
      strcpy(segundo_astro, segundo);
    }
 };
```

```
void main(void)
 {
   filme fugitivo, feitico;

   fugitivo.inicializa("O Fugitivo", "Harrison Ford", "Tommy Lee Jones");
   feitico.inicializa("O Feitiço da Lua", "Tom Hanks",  "Meg Ryan");
   cout << "Os dois últimos filmes a que assisti foram: "
        << fugitivo.nome  << " e " << feitico.nome << endl;
   cout << "Achei que o " << fugitivo.primeiro_astro << " estava ótimo!"
   << endl;
 }
```

916 DETERMINANDO QUANDO USAR FUNÇÕES IN-LINE OU NÃO

Foi visto na dica anterior que quando você declara uma função-membro in-line, os comandos da função residem dentro da própria classe. Uma vantagem de declarar funções-membro in-line é que elas ajudam a consolidar a classe inteira em uma posição dentro do código do seu programa. Infelizmente, usar funções in-line desse modo também aumenta o tamanho e a complexidade das suas definições de classe. Basicamente, quanto maiores se tornarem suas definições de classe, mais difíceis de compreender serão suas definições. Além disso, os tipos de objetos similares não compartilham o código para as funções in-line.

Por outro lado, quando você define funções-membro fora de uma classe, o compilador C++ cria uma cópia das instruções de cada função. Todo objeto que seus programas criarem mais tarde a partir dessa classe usa a cópia simples da função. Em outras palavras, se você criar mil objetos, cada objeto compartilha a única cópia do código da função. Esse compartilhamento de função é desejável porque reduz significativamente a utilização de memória do seu programa.

917 COMPREENDENDO AS CLASSES E AS UNIÕES

Como você aprendeu, as estruturas de C++ são essencialmente classes de C++. Do mesmo modo, você também pode usar as uniões de C++ para declarar uma classe. As uniões também podem incluir funções construtoras e destrutoras. Uma união em C++ retém todos seus aspectos similares a C (sobre os quais você aprendeu nas Dicas 481 até 487), incluindo a característica que força todos os elementos de dados à mesma posição na memória. Como uma estrutura, e ao contrário de uma classe, os membros da união são públicos por padrão. Como você aprendeu, um dos melhores usos para as uniões é manipular números usando operações bit a bit. O programa a seguir, *un_class.cpp*, usa a união *permuta_byte* para manipular os membros usando operações bit a bit:

```
#include <iostream.h>

union permuta_byte
{
   void permuta(void);
   void define_byte(unsigned i);
   void exibe_word(void);

   unsigned u;
   unsigned char c[2];
};

void permuta_byte::permuta()
{
   unsigned char t;

   t = c[0];
   c[0] = c[1];
   c[1] = t;
}

void permuta_byte::exibe_word()
```

```
{
   cout << u;
 }

void permuta_byte::define_byte(unsigned i)
 {
   u = i;
 }

void main(void)
 {
   permuta_byte b;

   b.define_byte(49034);
   b.permuta();
   b.exibe_word();
 }
```

INTRODUZINDO AS UNIÕES ANÔNIMAS 918

C++ suporta um tipo especial de união chamado *união anônima*. Uma união anônima não contém um nome de tipo e o programa não pode declarar variáveis a partir de uma união anônima. Em vez disso, uma união anônima diz ao compilador que as variáveis-membro da união devem compartilhar a mesma posição. No entanto, o programa referenciará as variáveis diretamente, sem a sintaxe do operador ponto normal. Para compreender melhor as uniões anônimas, considere o programa *uni_anon.cpp*, como mostrado aqui:

```
#include <iostream.h>
#include <string.h>

void main(void)
 {
   // define união anônima
   union
    {
      long l;
      double d;
      char s[4];
    };

   // agora, seu programa pode referenciar os elementos diretamente

   l = 100000;
   cout << l << " ";
   d = 123.2342;
   cout << d << " ";
   strcpy(s, "Oi");
   cout << s;
 }
```

INTRODUZINDO AS FUNÇÕES AMIGAS 919

É possível conceder a uma função não-membro acesso aos membros privados de uma classe. Para fazer isso dentro de seus programas, você pode declarar uma função *amiga*. As funções amigas têm acesso a todos os membros privados e protegidos da classe da qual são amigas. Para declarar uma função amiga, inclua seu protótipo dentro da classe, precedendo-a com a palavra-chave *friend*, como mostrado no programa *amiga.cpp:*

```
#include <iostream.h>

class exemplo
 {
```

```
    int a, b;
    public:
       friend int soma(exemplo x);
       void define_ab(int i, int j);
  }

void exemplo::define_ab(int i, int j)
  {
    a = i;
    b = j;
  }

int soma(exemplo object)
  {
    // Como soma é uma amiga de exemplo, ela pode acessar
    // a e b diretamente
    return object.a + object.b;
  }

void main(void)
  {
    exemplo integer;

    cout << "Somando 3 e 4:" << endl;
    integer.define_ab(3,4);
    cout << "Resultado = " soma(integer);
  }
```

Quando você compilar e executar o programa *amiga.cpp*, ele exibirá o seguinte resultado:

```
Somando 3 e 4:
Resultado = 7
C:\>
```

Embora, nesse caso, não haja uma razão particular para tornar *soma* uma função amiga e não uma função membro, existem geralmente três boas razões para usar uma função amiga dentro de suas classes:

1. As amigas podem ser úteis quando seu programa sobrecarrega certos operadores por causa do controle adicional que as funções amigas lhe dão sobre as ações do operador sobrecarregadas.
2. As funções amigas facilitam a criação de algumas funções de E/S.
3. As funções amigas também podem ser úteis em casos em que duas ou mais classes contêm membros inter-relacionados relativos a outras partes do programa, pois podem lhe ajudar a declarar múltiplas funções com código de programa igual.

Em dicas posteriores você aprenderá mais sobre como usar as funções amigas em cada um desses casos.

920 INTRODUZINDO AS CLASSES AMIGAS

Como você aprendeu, uma classe pode conter dados e métodos públicos e privados. Normalmente, o único modo de acessar os membros privados é por meio de métodos de interface ou públicos. À medida que seus programas começarem a trabalhar com mais de um tipo de objeto, algumas vezes um objeto chamará outro objeto ou usará os membros de dados de outro objeto. Por exemplo, em dicas anteriores, o objeto *Leitor* usou o método do objeto *Livro exibe_livro* para mostrar o título de um livro. O único modo de o objeto *Leitor* poder acessar os dados privados do objeto *Livro* era por meio do método *exibe_livro*. Dependendo do seu programa, algumas vezes você desejará que um objeto tenha acesso aos dados públicos e privados de outro objeto. Nesses casos, você pode especificar um objeto amigo. Dado o programa *Leitor* e *Livro* anterior, o objeto *Livro* poderia declarar o objeto *Leitor* como um amigo. O objeto *Leitor* poderia então acessar diretamente os dados privados do objeto *Livro*, exibindo o título do livro sem ter que chamar o método *exibe_titulo*. O restante do código do programa poderia não acessar diretamente os dados privados do objeto *Livro*. O único objeto que poderia acessar os dados privados seria o amigo do objeto *Livro*, o objeto *Leitor*. No entanto, antes de você especificar um amigo, deverá informar o compilador sobre a classe do amigo, como irão detalhar dicas posteriores.

COMPREENDENDO AS FUNÇÕES CONSTRUTORAS 921

Quando seu programa cria a ocorrência de um objeto, o programa então normalmente atribui valores iniciais aos membros de dados do objeto. Para simplificar o processo de inicializar membros do objeto, C++ suporta uma função especial, chamada *construtora*, que executa automaticamente cada vez que seu programa cria a ocorrência de uma classe. A função construtora é um método público que usa o mesmo nome que a classe. Por exemplo, usando a classe *Livro*, a função construtora teria o nome *Livro*, como mostrado aqui:

```
class Livro
 {
  public:
    Livro(char *titulo, char *autor, char *editora, float preco);    //
    Construtora
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void) { cout << titulo << \n'; };
    float pega_preco(void)  { return(preco); };
    void exibe_livro(void)
    {
       exibe_titulo();
       exibe editora();
    };
    void assign_editora(char *nome) { strcpy(editora, nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora <<'\n'; };
 };
```

Seus programas podem definir a função construtora dentro da própria classe ou fora dela. Quando seu programa, mais tarde, declarar um objeto, o programa poderá passar parâmetros para a função construtora. Em seguida, a função construtora executará automaticamente. Você pode passar parâmetros para a construtora, como mostrado a seguir:

```
Livro dicas("Bíblia do Programador C/C++", "Jamsa e Klander", "Jamsa
            Press", 49.95);
```

A Dica 922 apresentará um programa que usa uma função construtora para inicializar ocorrências da classe *Livro*.

USANDO FUNÇÕES CONSTRUTORAS COM PARÂMETROS 922

Na dica anterior, você aprendeu que um programa pode passar parâmetros para a função construtora. Também é possível passar argumentos para as funções construtoras. Tipicamente, seus programas usarão esses argumentos para ajudar a inicializar um objeto que o programa criar. Para criar uma *construtora parametrizada*, você simplesmente adiciona parâmetros à declaração da construtora, como faria com qualquer outra função. Ao definir o corpo da construtora, use os parâmetros para inicializar o objeto. Por exemplo, a declaração de classe a seguir inicializa a classe *Livro* dentro da construtora da classe:

```
class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco; void Livro(char *titulo, char *autor, char *editora, float
    preco);
    void exibe_titulo(void) { cout << titulo << '\n'; };
    float pega_preco(void)  { return(preco); };
    void exibe_livro(void)
    {
       exibe_titulo();
```

```
      exibe_editora();
    };
    void assign_editora(char *nome) { strcpy(editora, nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n'; };
};
```

Se você não usar uma função construtora parametrizada dentro de uma classe, poderá sempre inicializar os valores dentro da classe após o programa construir o objeto.

923 USANDO UMA FUNÇÃO CONSTRUTORA

Já foi visto que uma função construtora é uma função de classe especial que executa automaticamente quando você cria a ocorrência de uma classe. Os programas normalmente usam funções construtoras para inicializar valores-membro. O programa a seguir, *construi.cpp*, usa a função construtora *Livro* para inicializar membros de ocorrências da classe *Livro:*

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    Livro(char *ltitulo, char *lautor, char *leditora, float lpreco);
    void exibe_titulo(void) { cout << titulo << '\n'; };
    float pega_preco(void)  { return(preco); };
    void exibe_livro(void)
    {
       exibe_titulo();
       exibe_editora();
    };
    void assign_editora(char *nome) { strcpy(editora, nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora <<'\n'; };
};

Livro::Livro(char *ltitulo, char *lautor, char *leditora, float lpreco)
 {
   strcpy(titulo, ltitulo);
   strcpy(autor, lautor);
   strcpy(editora, leditora);
   preco = lpreco;
 }

void main(void)
 {
   Livro dicas("Bíblia do Programador C/C++, do Jamsa!",
               "Jamsa e Klander", "Makron Books", 49.95);
   Livro diario("Todos os Meus Segredos...", "Kris Jamsa",
                "Nenhuma", 9.95);
   dicas.exibe_livro();
   diario.exibe_livro();
 }
```

No programa *constru.cpp*, a função construtora *Livro* precede cada um de seus nomes de parâmetros com a letra *l* para distinguir os nomes dos membros da classe. No entanto, como você aprendeu, seus programas podem preceder os nomes de variáveis com o nome de classe para resolver os conflitos de nomes.

COMPREENDENDO QUANDO UM PROGRAMA EXECUTA UMA CONSTRUTORA 924

Como regra geral, o compilador chamará a construtora de um objeto quando o código do programa declarar o objeto. No entanto, a hora real quando o compilador chama o código da construtora pode variar dependendo do tipo da classe e da posição dentro do projeto. Por exemplo, a função construtora de um objeto local executa quando o contador de programa encontra o comando de declaração do objeto. Além disso, se seu programa cria dois ou mais objetos dentro do mesmo comando, o programa executará as funções construtoras para cada objeto na ordem de declaração, da esquerda para a direita.

Para os objetos globais, o programa executa a função construtora antes que *main* inicie sua execução. Exatamente como com os objetos locais, as construtoras globais executam em ordem, da esquerda para a direita, e do alto para baixo. É impossível saber a ordem de execução para uma série de construtoras globais dispersas em múltiplos arquivos de código-fonte. O programa a seguir, *exib_con.cpp*, exibe a execução das funções construtoras:

```
#include <iostream.h>

class exemplo
{
public:
   int quem;
   exemplo(int id);
 } global_obj1(1), global_obj2(2);

exemplo::exemplo(int id)
{
   cout << "Inicializando " << id << "\n";
   quem = id;
 }

void main(void)
{
   exemplo local_obj(3);
   cout << "Esta NÃO é a primeira linha mostrada.\n";
   exemplo local_obj2(4);
 }
```

Nota: Uma função construtora é uma função de classe especial que executa automaticamente quando o programa cria uma ocorrência. As funções construtoras não retornam um valor. No entanto, você não define funções construtoras como retornando o tipo void. Em vez disso, o compilador C++ pode determinar que uma função é uma construtora pela maneira como você usa a função. Por definição, não é possível retornar um valor de uma construtora.

USANDO FUNÇÕES CONSTRUTORAS COM PARÂMETROS 925

Como você aprendeu na Dica 921, é possível passar argumentos para as funções construtoras. Tipicamente, seus programas usarão esses argumentos para ajudar a inicializar um objeto quando o programa o criar. Para criar uma construtora parametrizada, você acrescenta parâmetros à declaração de uma construtora como faria com qualquer outra função do Windows. Quando você definir o corpo de uma construtora, use os parâmetros para inicializar os objetos. Por exemplo, a declaração de classe a seguir inicializa os objetos da classe *Livro*:

```
Livro::Livro (char *ltitulo, char *lautor, char *leditora, float lpreco)
  {
     strcpy (titulo, ltitulo);
     strcpy (autor, lautor);
     strcpy (editora, leditora);
     preco = lpreco;
 }
```

Se a função construtora dentro do exemplo anterior for a única função construtora que você criou para o objeto *livro*, seus programas precisarão declarar cada ocorrência com valores dentro da declaração que corresponda aos parâmetros *ltitulo*, *lautor*, *leditora* e *lpreco*. Caso deixe de fazer isso, o compilador retornará um erro.

926 RESOLVENDO OS CONFLITOS DE NOMES EM UMA FUNÇÃO CONSTRUTORA

Nas dicas anteriores, você criou a função construtora *Livro*, depois a modificou para inicializar os membros para ocorrências da classe *Livro*. Para diferenciar entre parâmetros e nomes de membro de classe, o programa precedeu cada nome do parâmetro com a letra *l*, como mostrado aqui:

```
Livro::Livro (char *ltitulo, char *lauthor,
              char *leditora, float lpreco).
 {
     strcpy (titulo, ltitulo);
     strcpy (autor, lautor);
     strcpy (editora, leditora);
     preco = lpreco;
 }
```

Nesse caso, os nomes dos parâmetros *ntulo*, *autor*, *editora* e *preco* são mais significativos e preferíveis que *ltitulo*, *leditora* e *lpreco*. No entanto, como os nomes dos parâmetros sem a inicial *l* entram em conflito com os nomes dos membros, a função precisa resolvê-los usando o nome da classe e dois-pontos dobrados, como a seguir:

```
Livro::Livro (char *titulo, char *autor, char *editora, float preco)
 {
    strcpy (Livro::titulo, titulo);
    strcpy (Livro::autor, autor);
    strcpy (Livro::editora, editora);
    Livro::preco = preco;
 }
```

927 USANDO UMA CONSTRUTORA PARA ALOCAR MEMÓRIA

Como você aprendeu, as funções construtoras permitem que seus programas inicializem as variáveis-membro. Se a variável-membro usa matrizes, da função construtora pode alocar a quantidade de memória que você quer. Por exemplo, o programa a seguir, *cons_new.cpp*, usa o operador *new* dentro da função construtora *Livro* para alocar memória para as matrizes de string de caracteres:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>

class Livro {
  public:
    char *titulo;
    char *autor;
    float preco;
    Livro(char *titulo, char *autor, char *editora, float preco);
    void exibe_titulo(void) { cout << titulo << '\n'; };
    float pega_preco(void)  { return(preco); };
    void exibe_livro(void)
    {
       exibe titulo();
       exibe_editora();
    };
    void atribui_editora(char *nome) { strcpy(editora, nome); };
```

```
  private:
    char *editora;
    void exibe_editora(void) { cout << editora << '\n';};
};

Livro::Livro(char *titulo, char *autor, char *editora, float preco)
 {
   if ((Livro::titulo = new char[256]) == 0)
     {
       cerr << "Erro ao alocar a memória\n";
       exit(0);
     }

   if ((Livro::autor = new char[64]) == 0)
     {
       cerr << "Erro ao alocar a memória\n";
       exit(0);
     }
   if ((Livro::editora = new char[128]) == 0)
     {
       cerr << "Erro ao alocar memória\n";
       exit(0);
     }

   strcpy(Livro::titulo, titulo);
   strcpy(Livro::autor, autor);
   strcpy(Livro::editora, editora);
   Livro::preco = preco;
 }

void main(void)
 {
   Livro dicas("Bíblia do Programador C/C++",
               "Jamsa e Klander", "Makron Books", 49.95);
   Livro diario("Todos os Meus Segredos...",
                "Kris Jamsa", "Nenhuma", 9.95);

   dicas.exibe_livro();
   diario.exibe_livro();
 }
```

TRATANDO A ALOCAÇÃO DE MEMÓRIA DE UM MODO MAIS CLARO 928

Na dica anterior, o programa *cons_nov.cpp* usou o operador *new* dentro de uma função construtora para alocar memória para os membros de string de caracteres. O código para cada alocação de memória foi similar, como mostrado aqui:

```
   if ((Livro::titulo = new char[256]) == 0)
     {
       cerr << "Erro ao alocar a memória\n";
       exit(0);
     }

   if ((Livro::autor = new char[64]) == 0)
     {
       cerr << "Erro ao alocar a memória\n";
       exit(0);
     }
```

```
if ((Livro::editora = new char[128]) == 0)
  {
    cerr << "Erro ao alocar memória\n";
    exit(0);
  }
```

Um modo de reduzir a quantidade de código duplicado é tentar alocar memória para cada variável, e, depois, testar após a última alocação para ver se as alocações foram bem-sucedidas, como a seguir:

```
Livro::titulo = new char[256];
Livro::autor = new char[64];
Livro::editora = new char[128];
if ((Livro::titulo && Livro::autor && Livro::editora)
    == 0)
  {
    cout << "Erro ao alocar memória\n";
    exit(1);
  }
```

Um segundo modo de reduzir o código é primeiro atribuir uma rotina de tratamento personalizada que exibirá a mensagem de erro e sairá quando o espaço livre puder satisfazer a solicitação. Você aprendeu como atribuir uma rotina de tratamento personalizada anteriormente, na Dica 871, que discutiu em detalhes a função *set_new_handler*.

929 VALORES DE PARÂMETRO PADRÃO PARA AS CONSTRUTORAS

Uma função construtora é um método de classe especial que executa automaticamente quando seu programa cria uma ocorrência de um objeto. Como você aprendeu, C++ permite que seu programa forneça valores padrão para os parâmetros das funções. As funções construtoras não são exceções. O programa a seguir, *def_cons.cpp*, usa os valores padrão 1, 2 e 3 para os membros da classe *NumerosMagicos:*

```
#include <iostream.h>
#include <iomanip.h>

class NumerosMagicos {
  public:
    NumerosMagicos(int a = 1, int b = 2, int c = 3)
      {
        NumerosMagicos::a = a;
        NumerosMagicos::b = b;
        NumerosMagicos::c = c;
      };
    void exibe_numeros(void)
      {
        cout << a << ' ' << b << ' '<< c << '\n';
      };
  private:
    int a, b, c;
};

void main(void)
 {
   NumerosMagicos um(1, 1, 1);
   NumerosMagicos padroes;
   NumerosMagicos happy(101, 101, 101);

   um.exibe_numeros();
   padroes.exibe_numeros();
   happy.exibe_numeros();
 }
```

Como você pode ver, as ocorrências *um* e *happy* especificam seus próprios valores-membro. A ocorrência chamada *padroes*, no entanto, usa os valores padrão 1, 2 e 3. Fornecendo valores padrão para sua função construtora desse modo, você pode garantir que seu programa sempre inicialize os membros de classes com valores significativos.

SOBRECARREGANDO AS FUNÇÕES CONSTRUTORAS 930

Uma função construtora é um método de classe especial que executa automaticamente quando seu programa cria uma ocorrência de um objeto. Como você aprendeu, C++ permite que seus programas sobrecarreguem funções para que o compilador C decida quais funções chamar, dependendo dos parâmetros passados. As funções construtoras não são exceções. O programa a seguir, *valneces.cpp*, fornece duas funções construtoras para a classe *Livro*. A primeira função construtora atribui os valores passados como parâmetros. A segunda construtora exibe primeiro uma mensagem dizendo que o programa precisa fornecer valores iniciais para cada parâmetro e depois termina. No programa *valneces.cpp*, a segunda construtora executa somente se o programa tentar executar uma função sem especificar valores iniciais, como mostrado aqui:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdlib.h>

class Livros {
  public:
    Livros(char *titulo, char *autor, char *editora, float preco);
    Livros(void);
    void exibe_titulo(void) { cout << titulo << '\n'; };
    float pega_preco(void) { return(preco); };
    void exibe_Livros(void)
    {
       exibe_titulo();
       exibe editora();
    };
  private:
    char titulo[256];
    char autor[64]; float preco;
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n';};
 };

Livros::Livros(char *titulo, char *autor, char *editora, float preco)
 {
   strcpy(Livros::titulo, titulo);
   strcpy(Livros::autor, autor);
   strcpy(Livros::editora, editora);
   Livros::preco = preco;
 }

Livros::Livros(void)
 {
   cerr << "Você precisa especificar os valores iniciais"
        << " para a ocorrência Livros\n";
   exit(1);
 }
void main(void)
 {
   Livros dicas("Bíblia do Programador C/C++", "Jamsa e Klander",
"Jamsa Press", 49.95);
   Livros diario;
```

```
   dicas.exibe_Livros();
   diario.exibe_Livros();
 }
```

931 ENCONTRANDO O ENDEREÇO DE UMA FUNÇÃO SOBRECARREGADA

Você já sabe que pode atribuir o endereço de uma função a um ponteiro e depois chamar esse ponteiro para chamar a função. No entanto, ao sobrecarregar funções dentro de seus programas, obter o endereço de uma função é mais complexo. Para compreender melhor por que obter o endereço de uma função é mais difícil com as funções sobrecarregadas, considere o comando a seguir, que atribui o endereço da função *algumaf* a um ponteiro chamado *p:*

```
p = algumaf;
```

Se a função *algumaf* não tiver funções sobrecarregadas, o comando anterior sozinho é suficiente para a atribuição do endereço. Por outro lado, se existem múltiplas funções *algumaf,* o compilador se recusará a compilar porque não pode resolver a referência. A resposta a esse problema está na declaração da própria variável, como mostrado no programa a seguir, *sobre_pt.cpp:*

```
#include <iostream.h>

int algumaf(int a);
int algumaf(int a, int b);

void main(void)
 {
   int (*pf)(int a);   // ponteiro para int xxx(int)

   pf = algumaf;       // aponta para algumaf(int)
   cout << pf(5);
 }

int algumaf(int a)
{
   return a;
 }

int algumaf(int a, int b)
{
   return a*b;
 }
```

Como o programa declara *pf* como um ponteiro para uma função que retorna um valor *int* e recebe um único parâmetro *int,* o compilador pode resolver o ponteiro quando outro comando do programa referenciar o ponteiro dentro do programa. Por outro lado, se a declaração *pf* fosse como a seguir, o ponteiro apontaria para a segunda função:

```
int (*pf)(int a, int b);
```

932 USANDO FUNÇÕES CONSTRUTORAS COM UM ÚNICO PARÂMETRO

Nas dicas anteriores você aprendeu como parametrizar as funções construtoras para inicializar os dados-membro de uma classe cada vez que o programa criar uma ocorrência dessa classe. No entanto, seus programas também podem manipular as funções construtoras com somente um único parâmetro, como se a declaração da classe fosse uma atribuição normal para um tipo. Para compreender melhor como seus programas implementarão as funções construtoras com um único parâmetro, considere o programa *unic_par.cpp,* mostrado aqui:

```
#include <iostream.h>

class simples
{
   int a;
   public:
     simples(int j)  {//a = j;//}
     int pega_a(void)  {return a;}
 );

void main(void)
{
   simples ob = 99;       // passa 99 para j

   cout << ob.pega_a();
 }
```

COMPREENDENDO AS FUNÇÕES DESTRUTORAS 933

Cada vez que você cria uma ocorrência de objeto, seu programa pode executar automaticamente uma função construtora que você pode usar para inicializar os membros da ocorrência. De um modo similar, C++ lhe permite definir uma função destrutora que roda automaticamente quando o programa destrói a ocorrência. As funções destrutoras tipicamente rodarão em uma das duas seguintes situações: quando seu programa terminar ou quando você usar o operador *delete* para liberar memória alocada anteriormente para conter uma ocorrência. As funções destrutoras têm o mesmo nome que a classe. Você diferenciará as destrutoras das construtoras pelo sinal de til (~), que precisa preceder o nome de toda função destrutora. Por exemplo, o fragmento de código a seguir mostra as declarações para a construtora e a destrutora de uma função:

```
Livro(char *titulo, char *autor, char *editora, float preco);
~Livro(void);
```

Como você pode ver, uma função destrutora não aceita parâmetros e, similar às funções construtoras, você declarará a função destrutora sem um valor de retorno. Você aprenderá mais sobre as funções destrutoras nas dicas a seguir.

USANDO UMA FUNÇÃO DESTRUTORA 934

Na dica anterior, você aprendeu sobre a contrapartida da função construtora, a destrutora. C++ chama automaticamente a função destrutora cada vez que um programa destrói uma ocorrência de classe. Para compreender melhor esse processamento, considere o programa a seguir, *destru.cpp*, que cria uma função destrutora simples que exibe uma mensagem dizendo que o programa está destruindo uma ocorrência. O programa chama automaticamente a função destrutora para cada ocorrência quando o programa termina, como mostrado aqui:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    Livro(char *titulo, char *autor, char *editora, float preco);
    ~Livro(void);
    void exibe_titulo(void) {cout << titulo << '\n';};
    float obtem_preco(void) {return(preco); };
    void exibe_Livro(void)
    {
       exibe_titulo();
       exibe_editora();
    };
```

```
    void assign_editora(char *nome) { strcpy(editora, nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n';};
};

Livro::Livro(char *titulo, char *autor, char *editora, float preco)
{
  strcpy(Livro::titulo, titulo);
  strcpy(Livro::autor, autor);
  strcpy(Livro::editora, editora);
  Livro::preco = preco;
}

Livro::~Livro(void)
{
  cout << "Destruindo a ocorrência " << titulo << '\n';
}

void main(void)
{
  Livro dicas("Bíblia do Programador C/C++",
       "Jamsa e Klander", "Makron Books", 49.95);
  Livro diario("Todos os Meus Segredos...",
               "Kris Jamsa", "Nenhuma", 9.95);

  dicas.exibe_Livro();
  diario.exibe_Livro();
}
```

935 COMPREENDENDO POR QUE VOCÊ DEVE USAR AS FUNÇÕES DESTRUTORAS

As Dicas 933 e 934, mostraram as funções destrutoras. Como você aprendeu, C++ automaticamente chama a função destrutora cada vez que seus programas descartam uma ocorrência de classe, exatamente como ele chama automaticamente a função construtora cada vez que seus programas criam uma ocorrência de classe. Em muitos casos, você verá que a função destrutora não efetua processamento especial. No entanto, à medida que seus programas forem se tornando mais complexos, você encontrará duas situações gerais em que uma classe precisa ter uma função destrutora.

Como regra geral, as funções destrutoras são mais importantes para a estrutura do seu programa quando suas classes alocam memória dinamicamente Se sua classe cria todos os membros de dados, matrizes, estruturas e assim por diante na criação de cada ocorrência, C++ automaticamente tratará a maior parte do processamento da destruição. Por outro lado, se seus objetos alocam memória (por exemplo, um objeto de lista ligada), você deve se certificar de que seu programa libera essa memória (um processo que os programadores chamam de *coleta de lixo*).

Adicionalmente, se seu programa usa uma série de objetos vinculados, a função destrutora lhe ajudará a manter a lista após a destruição de um objeto dentro da lista. Como você aprendeu, em uma lista ligada simples, cada objeto mantém o endereço do próximo objeto na lista. Em uma lista duplamente ligada, cada objeto mantém o endereço do próximo objeto e do anterior na lista. Cada vez que você exclui um elemento (um nodo) de dentro da lista ligada, seu programa precisa atualizar os elos dentro da lista para evitar que ela fique interrompida. Como você aprenderá em dicas posteriores, é possível efetuar a maior parte das tarefas de limpeza doméstica em uma lista ligada dentro da função destrutora.

936 COMPREENDENDO QUANDO UM PROGRAMA CHAMA UMA FUNÇÃO DESTRUTORA

Ao criar funções destrutoras para seus programas, você deverá compreender quando seus programas invocarão as funções destrutoras que você criar para suas classes. Basicamente, C++ chama a função destrutora do objeto

imediatamente antes de descartar o objeto. Para compreender melhor como as funções destrutoras funcionam, considere a Figura 936, um esquema lógico simples do ciclo de vida de um objeto.

Figura 936 Modelo lógico da vida de um objeto.

Portanto, dentro de seus programas, você deve garantir que suas funções destrutoras executem somente atividades apropriadas no objeto prestes a ser destruído e que seus programas não planejem a execução da função destrutora antes do final da vida de um determinado objeto. Para compreender melhor o conceito dos ciclos de vida do objeto, considere o programa a seguir, *pilha_cd.cpp*, que constrói e depois destrói alguns objetos do tipo *pilha:*

```
#include <iostream.h>
#define TAM_MATRIZ 100

class pilha {
   int plh[TAM_MATRIZ];
   int pilha_topo;
 public:
    pilha();
   ~pilha();
   void push(int i);
   int pop();
 };

pilha::pilha(void)
 {
   pilha_topo = 0;
   cout << "Pilha Inicializada" << endl;
 }

pilha::~pilha(void)
 {
   cout << "Pilha Destruída" << endl;
 }

void pilha::push(int i)
 {
    if (pilha_topo==TAM_MATRIZ)
    {
       cout << "A pilha está cheia." << endl;
      return;
    }
   plh[pilha_topo] = i;
   pilha_topo++;
 }

int pilha::pop(void)
 {
```

```
  if (pilha_topo==0)
    {
      cout << "Estouro da pilha." << endl;
      return 0;
    }
  pilha_topo--;
  return plh[pilha_topo];
}

void main(void)
{
  pilha obj1, obj2;

  obj1.push(1);
  obj2.push(2);
  obj1.push(3);
  obj2.push(4);
  cout << obj1.pop() << endl;
  cout << obj1.pop() << endl;
  cout << obj2.pop() << endl;
  cout << obj2.pop() << endl;
}
```

Quando você compilar e executar o programa *pilha_cd.cpp*, verá que ele chama automaticamente a função destrutora para os dois objetos *pilha* imediatamente antes de o programa completar a execução. Quando você trabalhar com matrizes de objetos de classe, lembre-se de que seu programa disparará a função destrutora para cada elemento na matriz de classe que o programa destruir. Em outras palavras, se você tiver uma matriz de 100 elementos da classe, quando seu programa destruir a matriz, ele chamará a função destrutora 100 vezes — uma vez para cada elemento. Você aprenderá mais sobre a manipulação de matrizes de classe em dicas posteriores.

937 USANDO UMA CONSTRUTORA DE CÓPIA

Por padrão, quando C++ copia um objeto, ele efetua uma *cópia bit a bit*, o que significa que o novo objeto é uma cópia exata do objeto original. No entanto, em alguns casos, uma cópia bit a bit pode causar mais problemas do que resolver. Por exemplo, se uma função recebe a ocorrência de um objeto por valor, depois cria uma cópia local da ocorrência dentro da função, quando o programa terminar a função, ele excluirá a cópia local do objeto, como você esperaria. No entanto, quando C++ exclui a cópia local, ele também excluirá a memória que a cópia de fora usa. Você evita problemas dessa natureza escrevendo uma função *construtora de cópia*. A forma geral da função construtora de cópia é como mostrado aqui:

```
nomeclasse (const nomeclasse &objeto)
{
  // Corpo da construtora
}
```

Neste caso, o parâmetro *objeto* é a ocorrência do objeto que você está copiando. Você também pode incluir parâmetros de inicialização dentro de uma construtora de cópia, embora tenha que fornecer valores padrão para cada parâmetro. O programa a seguir, *cons_cop.cpp*, usa uma construtora de cópia dentro de uma classe *matriz*:

```
#include <iostream.h>
#include <stdlib.h>

class matriz {
  int *p;
  int tamanho;
public:
  matriz(int tm)   {          // construtora simples
    p = new int[tm];
    if(!p) exit(1);
    tamanho = tm;
  }
```

```
   ~matriz() {delete [] p;}      // destrutora
   matriz(const matriz &object);  // cópia construtora
   void put(int i, int j) {
      if(i>=0 && i<tamanho)
        p[i] = j;
    }
   int get(int i) {return p[i];}
};

matriz::matriz(const matriz &object)
{
   int lcl_i;

   p = new int[object.tamanho];
   if (!p)
      exit(1);
   for(lcl_i=0; lcl_i < object.tamanho; lcl_i++)
      p[lcl_i] = object.p[lcl_i];
}

void main(void)
{
   matriz num(10);
   int lcl_i;

   for (lcl_i=0; lcl_i<10; lcl_i++)
      num.put(lcl_i, lcl_i);
   for (lcl_i=9; lcl_i>=0; lcl_i--)
      cout << num.get(lcl_i);
   cout << endl;

   //  Cria outra matriz usando a construtora de cópia
   matriz x=num;
   for (lcl_i=0; lcl_i<10; lcl_i++)
      cout << x.get(lcl_i);
}
```

Quando o programa *cons_cop.cpp* é executado, ele cria primeiro o objeto *num*, que é inicializado e exibido. Em seguida, o programa usa *num* para inicializar *x*, chamando a construtora de cópia no processo. A construtora de cópia copia todos os dados dentro do objeto *num* para o objeto *x*, mas, ao fazer isso, cria o próprio espaço de memória da cópia de *x*, independente do objeto *num*.

Nota: Seus programas somente podem chamar uma construtora de cópia durante as inicializações. Se seus programas tiverem criado um objeto e depois tentarem copiar outro objeto para esse objeto, a construtora da cópia não interferirá.

USANDO CONSTRUTORAS EXPLÍCITAS 938

Você pode usar *construtoras explícitas* dentro de seus programas para forçar todas as declarações para a forma que sua construtora determina explicitamente. Em geral, quando você cria uma construtora, como mostra o fragmento de código a seguir, múltiplos estilos de inicialização são aceitáveis para o compilador:

```
exemploclasse (int j) {I=j;}
//
// Comandos do programa aqui

exemploclasse obj1(10);
exemploclasse obj2 = 10;
//
// Mais comandos do programa
```

No entanto, se você declarar a classe como explícita, o compilador permitirá que o programa use construtoras somente do tipo e formato especificados. Usar classes explícitas é provavelmente mais útil em bibliotecas

de classes e outras posições de classe semifixas. Você usará a palavra-chave *explicit* para definir uma classe, como a seguir:

```
explicit exemploclasse (int j)  {I=j;}
```

939 COMPREENDENDO O ESCOPO DA CLASSE

Como você aprendeu, o *escopo* de um identificador define as posições dentro do programa nas quais o identificador é conhecido. As classes de C++, como tipos e variáveis, têm um escopo que começa em suas definições dentro do arquivo de programa e existe até o final do bloco dentro do qual as classes foram definidas. Para aumentar o escopo de uma classe, você pode definir a classe fora de todos os blocos do programa. Além disso, se você definir uma classe como *extern*, a classe será conhecida em todo o programa. Se você definir uma classe como *static*, o escopo da classe permanecerá o mesmo como se a classe fosse automática, mas a existência da classe permanecerá durante a duração do programa.

940 COMPREENDENDO AS CLASSES INSERIDAS DENTRO DE OUTRAS

Já vimos em seções anteriores deste livro, que é possível definir uma estrutura dentro de outra estrutura. Da mesma forma, também é possível definir uma classe dentro de outra. Como a declaração de uma classe na verdade define um escopo, uma classe inserida dentro de outra somente é válida dentro do escopo da classe envolvente. Por essa razão, você raramente deve usar classes inseridas dentro de outras em seus programas. Dada a flexibilidade de C++, especialmente de seus mecanismos de herança (sobre os quais você aprenderá em dicas posteriores), não há realmente necessidade de usar classes inseridas.

941 COMPREENDENDO AS CLASSES LOCAIS

Exatamente como você pode definir variáveis locais dentro de uma função, assim também é possível definir uma classe dentro de uma função. Quando você declara uma classe dentro de uma função, a classe é conhecida somente dentro dessa função. O programa a seguir, *local_cl.cpp*, define uma classe local válida:

```
#include <iostream.h>

void f(void);
void main(void)
 {
   f();
 }

void f(void)
 {
   class minhaclasselocal
    {
      int i;
   public:
      void put_i(int n)  {i=n;}
      int get_i() {return i;}
    } ob;

   ob.put_i(10);
   cout << ob.get_i();
 }
```

C++ aplica várias restrições às classes locais que as tornam incomuns dentro dos programas C++, incluindo as seguintes:

1. Você precisa definir todas as funções-membro dentro da declaração da classe (em outras palavras, todas as funções-membro precisam ser *in-line*).
2. A classe local não pode usar ou acessar variáveis locais da função na qual você a declara
3. Você não pode declarar qualquer variável *estática* dentro de uma classe local.

RESOLVENDO OS CONFLITOS DE NOME DE MEMBRO E DE PARÂMETRO 942

Dentro de suas funções-membro, algumas vezes um nome de classe-membro está em conflito com o nome de um parâmetro passado para a função. Por padrão, C++ resolve esses conflitos de nomes usando o parâmetro (a variável local) e ocultando a existência do membro da classe. Para impedir esses conflitos de nomes, preceda as referências do nome da classe com o nome da classe e dois-pontos dobrados, como mostrado aqui:

```
void caes::atribui_cao(char *raca, int altura, int peso)
 {
   strcpy(caes::raca, raca);
   caes::altura = altura;
   caes::peso = peso;
 }
```

Neste caso, os nomes que *caes*:: precede correspondem aos nomes dos membros da classe. Os outros nomes correspondem às variáveis locais.

CRIANDO UMA MATRIZ DE VARIÁVEIS DE CLASSE 943

Várias dicas apresentadas neste livro criaram matrizes de estruturas. De um modo similar, seus programas podem criar uma matriz de ocorrências de classe. O programa a seguir, *biblio.cpp*, cria uma matriz que contém os detalhes específicos sobre quatro livros:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    void exibe_titulo(void)  { cout << titulo << '\n'; };
    void exibe_livro(void)
      {
      exibe_titulo();
      exibe_editora(); };
    void atribui_membros(char *, char *, char *, float);
  private:
    char titulo[256];
    char autor[64];
    float preco;
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n';};
 };

void Livro::atribui_membros(char *titulo, char *autor, char *editora, float
preco)
 {
   strcpy(Livro::titulo, titulo);
   strcpy(Livro::autor, autor);
   strcpy(Livro::editora, editora);
   Livro::preco = preco;
 }

void main(void)
```

```
{
  Livro Biblio[4];

  Biblio[0].atribui_membros ("Bíblia do Programador C/C++",
            "Jamsa e Klander", "Makron Books", 49.95);
  Biblio[1].atribui_membros("Banco de Dados com VB 5",
            "Ganz", "Makron Books", 54.95);
  Biblio[2].atribui membros("C++ Completo e Total",
       "Schildt", "Makron Books", 49.95);
  Biblio[3].atribui_membros("Aprendendo C++", "Jamsa",
       "Makron Books", 29.95);

  for (int i = 0; i < 4; i++)
     Biblio[i].exibe_livro();
}
```

944 CONSTRUTORAS E MATRIZES DE CLASSES

Como você aprendeu, C++ permite que seus programas declarem matrizes de um tipo de classe específico. Quando você declarar uma matriz, C++ automaticamente chamará a função construtora para cada entrada na matriz. Por exemplo, o programa a seguir, *classemt.cpp*, cria uma matriz do tipo de classe *Funcionario:*

```
#include <iostream.h>

class Funcionario
 {
 public:
   Funcionario(void) {  cout << "Construindo uma ocorrência\n";  };
   void exibe_funcionario(void)  { cout << nome; };

 private:
   char nome[256];
   long id;
};

void main(void)
 {
   Funcionario trabalhadores[5];
  // Outros comandos aqui
 }
```

Quando você compilar e executar o programa *classemtz.cpp*, verá que o programa chama automaticamente a função construtora cinco vezes, uma vez para cada elemento da matriz.

945 SOBRECARREGANDO UM OPERADOR

Você sabe que, ao sobrecarregar uma função, o compilador C determina qual função chamar, com base no número e tipos de parâmetros. Ao criar uma classe, C++ também lhe permite sobrecarregar os operadores. Quando você sobrecarrega um operador, precisa continuar a usar o operador em seu formato padrão. Por exemplo, se você sobrecarrega o operador de adição (+), a sobrecarga ainda precisa usar o operador na forma *operando + operando*. Além disso, você pode somente sobrecarregar os operadores existentes. C++ não permitirá que você defina seus próprios operadores. A sobrecarga que você cria se aplica somente às ocorrências da classe específica. Por exemplo, assuma que você crie uma classe *String* e sobrecarregue o sinal de adição para que o operador concatene duas strings, como a seguir:

```
nova_string = string + alvo;
```

Se você usar o operador de adição sobrecarregado com dois valores inteiros ou de ponto flutuante, a sobrecarga não se aplicará. Além disso, C++ não permitirá que você sobrecarregue os operadores listados na Tabela 945.

Tabela 945 Os operadores que C++ permite que você sobrecarregue.

Operador	Função
	Operador de membro da classe
.*	Operador ponteiro para membro
::	Operador de definição de escopo
?:	Operador de expressão condicional

CRIANDO UMA FUNÇÃO DE OPERADOR DE MEMBRO 946

Quando você criar funções de operador de membro para sobrecarregar a funcionalidade de um operador, as declarações de seu operador de membro tomarão a forma geral, como mostrado aqui:

```
tipo-retorno nome-classe::operator #(lista-argumento)
 {
   // Operações
 }
```

Normalmente, as funções *operator* retornam um objeto da classe em que operam. No entanto, C++ lhe permite definir *tipo-retorno* como qualquer tipo válido. O símbolo # representa um marcador de lugar para o operador que você quer sobrecarregar. Por exemplo, na dica a seguir você sobrecarregará o operador de adição usando uma declaração de função similar a esta mostrada aqui:

```
char *operator +(char *anexa_str)
```

Nesta declaração de função, o operador de membro é o sinal de adição. Quando você estiver sobrecarregando um operador unário (isto é, um operador que atua somente em um único valor), a *lista-argumento* precisará estar vazia. Quando você estiver sobrecarregando um operador binário, a *lista-argumento* precisará conter somente um único parâmetro.

A razão para essa construção aparentemente estranha é que C++ passa automaticamente o valor no lado esquerdo do operador para a função sobrecarregada. Portanto, quando você chama um operador unário, C++ automaticamente passa o valor com o qual a função está operando para a função sobrecarregada. Como você aprenderá mais tarde, compreender como C++ passa valores para as funções sobrecarregadas é especialmente importante quando você manipular os operadores de incremento e de decremento de prefixo e de sufixo.

SOBRECARREGANDO O OPERADOR DE ADIÇÃO 947

Você aprendeu que para sobrecarregar um operador, precisa criar uma classe na qual quer que a sobrecarga se aplique. Após você ter criado a classe, precisa colocar dentro dos métodos públicos para a classe uma linha de cabeçalho que define o operador. Por exemplo, o programa a seguir, *sobre_ms.cpp* cria uma classe *String* e sobrecarrega o sinal de adição (+) para que ele concatene strings:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class String
 {
  public:
    char *operator +(char *anexa_str)
       { return(strcat(buffer, anexa_str)); };

    char *operator -(char letra);

    String(char *string)
      { strcpy(buffer, string);
        comprimento = strlen(buffer); }

    void exibe_string() { cout << buffer; };
```

```
  private:
    char buffer[256];
    int comprimento;
};

void main(void)
{
  String titulo("Bíblia do Programador ");

  titulo = titulo + "C/C++, do Jamsa!\n";
  titulo.exibe_string();
}
```

Quando você rodar o programa *sobre_ms.cpp*, ele começará a atribuir ao membro *buffer* a string "Bíblia do Programador". O programa então usa o operador de mais sobrecarregado para concatenar os caracteres "C/C++, do Jamsa!". Observe que a sobrecarga do operador é simplesmente uma função que recebe um parâmetro. A função recebe somente um parâmetro. O parâmetro é o segundo operando. A própria operação implica o operando da ocorrência.

O operador de adição sobrecarregado usa as funçoes *strcpy* e *strcat* para copiar a string entre aspas para o objeto *titulo*. Observe que o código dentro da função de sobrecarga do operador de adição se refere aos dados-membro do objeto *titulo* implicitamente, com comandos tais como os seguintes, que colocam o valor atual do título no objeto *temp*:

```
strcpy(temp.buffer, buffer);
```

O programa poderia, da mesma forma, referenciar o objeto explicitamente, usando o ponteiro *this*, como mostrado aqui:

```
strcpy(temp.buffer, this.buffer);
```

948 SOBRECARREGANDO O OPERADOR DO SINAL DE SUBTRAÇÃO

Na dica anterior, você criou uma classe *String* e sobrecarregou o sinal de adição. O programa a seguir, *strmenos.cpp*, sobrecarrega o operador de subtração (-), depois usa o operador sobrecarregado para remover todas as ocorrências de um caractere especificado da classe-membro *buffer*:

```
 {/* Outro modo de sobrecarregar o operador -    */

#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class String
{
  public:
    char *operator +(char *anexa_str)
      { return(strcat(buffer, anexa_str)); };

    char *operator -(char letra);

    String(char *string)
      { strcpy(buffer, string);
        comprimento = strlen(buffer); }

    void exibe_string(void) { cout << buffer; };
  private:
    char buffer[256];
    int comprimento;
};

char *String::operator -(char letra)
```

```
{
   char alvo[256];
   int i, j;

   for (i = 0, j = 0; buffer[j]; j++)
     if (buffer[j] != letra)
       alvo[i++] = buffer[j];
   alvo[i] = NULL;

   for (i = 0, j = 0; (buffer[j] = alvo[i]); i++, j++)
     ;
   return(buffer);
 }

void main(void)
{
   String titulo("Bíblia do Programador ");

   titulo = titulo + "C/C++, do Jamsa!\n";
   titulo.exibe_string();
   titulo = titulo - '0';
   titulo.exibe_string();
 }
```

Quando você rodar o programa *strmenos.cpp*, ele começará atribuindo ao membro *buffer* a string "Bíblia do Programador". O programa então usa o operador de adição sobrecarregado para concatenar os caracteres "C/C++, do Jamsa!". Observe que a sobrecarga do operador é simplesmente uma função que recebe um parâmetro. A função recebe somente um parâmetro. O parâmetro é o segundo operando. A própria operação implica o operando da ocorrência.

O operador de subtração sobrecarregado usa um laço *for* simples para percorrer a matriz de caracteres um elemento de cada vez. Se o elemento não for igual à primeira letra da substring, ele copiará o elemento para o objeto *temp* e passará para o próximo elemento. Se o elemento não for igual à primeira letra da substring, ele entrará em um segundo laço, que verifica os elementos em relação aos elementos da substring. Se eles forem iguais, o processo de cópia pulará a substring inteira. Se eles não forem iguais, o processo de cópia reverterá para o elemento inicial e processará da posição da matriz *char*. Observe que o código dentro da função de sobrecarga do operador de subtração se refere aos membros de dados do objeto *titulo* implicitamente, com comandos tais como os seguintes, que colocam o valor atual do título no objeto *temp:*

```
s1 = buffer;
```

O programa poderia, da mesma forma, referenciar o objeto explicitamente, usando o ponteiro *this*, como a seguir:

```
s1 = this.buffer;
```

Sobrecarregando os Operadores de Incremento de Prefixo e de Sufixo 949

Como você aprendeu, é possível sobrecarregar operadores e funções em C++. Um dos conjuntos de operadores mais comumente usados em C++ é o dos operadores de prefixo e de sufixo. Como você aprendeu, se colocar o operador de incremento (++) antes de uma variável, C++ incrementará a variável antes de interpretá-la; se você colocar o operador após a variável, C++ interpretará a variável antes de incrementá-la.

As versões anteriores de C++ não forneceram ao programa um modo de sobrecarregar os operadores de prefixo e de sufixo separadamente. No entanto, as versões modernas de C++ fornecem um modo de determinar se um incremento prefixia ou sufixia seus operandos. Para sobrecarregar o incremento de prefixo ou de sufixo, você definirá duas versões da função *operator++*, como mostrado aqui:

```
nome-classe operator++();
nome-classe operator++(int x);
```

Se o operador de incremento preceder seu operando, o compilador chamará a função *operator++()*. Se, por outro lado, o operador de incremento seguir seu operando, o compilador chamará a função *operator++(int x)*. O programa a seguir, *sobre_in.cpp*, sobrecarrega o operador de incremento para a classe *string*:

```
// Este arquivo não compila no TCLite porque ele não
// suporta operadores de incremento duais sobrecarregados

#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class String
 {
  public:
    String String::operator++()
      { strcat(buffer, "X");
        return *this; };

    String String::operator++(int x)
      { strcat(buffer, "X");
        return *this; };

    String(char *string)
      { strcpy(buffer, string);
  comprimento = strlen(buffer); }

    void exibe_string(void) { cout << buffer << endl; };
  private:
    char buffer[256];
    int comprimento;
 };

void main(void)
 {
   String titulo
          ("Bíblia do Programador C/C++, do Jamsa!");

   titulo++;
   titulo.exibe_string();
   ++titulo;
   titulo.exibe_string();
 }
```

950 SOBRECARREGANDO OS OPERADORES DE DECREMENTO DE PREFIXO E DE SUFIXO

Vimos na dica anterior as versões modernas de C++ lhe oferecem um modo de sobrecarregar os tipos de prefixo e de sufixo de um determinado operador. Exatamente como você declarará duas funções de operadores para sobrecarregar o operador de incremento, assim também criará duas funções de operador para sobrecarregar o operador de decremento, como mostrado aqui:

```
nome-classe operator--();
nome-classe operator--(int x);
```

Se o operador de decremento preceder seu operando, o compilador chamará a função *operator--()*. Se, por outro lado, o operador de decremento seguir seu operando, o compilador chamará a função *operator--(int x)*. O programa a seguir, *sobre_dc.cpp*, sobrecarrega os operadores de decremento:

```
// Este arquivo não compila no TCLite porque ele não
```

```
// suporta os operadores de decremento sobrecarregados
// duais

#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class String
 {
  public:
    String String::operator--()
      { buffer[comprimento-1] = NULL;
        comprimento--;
        return *this; };

    String String::operator--(int x)
      { buffer[comprimento-1] = NULL;
        comprimento--;
        return *this; };

    String(char *string)
      { strcpy(buffer, string);
        comprimento = strlen(buffer); }

    void exibe_string(void) { cout << buffer << endl; };
  private:
    char buffer[256];
    int comprimento;
 };

void main(void)
 {
   String titulo ("Bíblia do Programador C/C++, do Jamsa!");

   titulo--;
   titulo.exibe_string();
   --titulo;
   titulo.exibe_string();
 }
```

REVISITANDO AS RESTRIÇÕES NA SOBRECARGA DE OPERADOR 951

Como você aprendeu anteriormente, C++ limita quais operadores seus programas podem sobrecarregar. Como vimos, não é possível sobrecarregar o operador *ponto*, o operador de *definição de escopo*, o operador de *condição* ou o operador de *indireção de ponteiro*. No entanto, além desses operadores, você pode livremente sobrecarregar qualquer operador que quiser.

Por exemplo, se você quiser sobrecarregar o operador de adição de tal forma que ele escreva "Happy é um dálmata" dez vezes na tela, pode fazer isso. No entanto, geralmente você não deve sobrecarregar um operador de um modo tão fundamentalmente diferente de seu uso normal. Quando outro programador ler seu código e vir *a + b*, esse programador deverá poder esperar que o operador de adiçao sobrecarregado efetue algum tipo de atividade de adição — não uma série de escritas na tela.

Exceto pelo operador de atribuição, as classes derivadas herdarão todos os operadores sobrecarregados a partir da classe-base. No entanto, suas classes derivadas permanecem livres para sobrecarregar qualquer operador elas próprias (até mesmo os operadores que a classe-base sobrecarrega).

952 USANDO UMA FUNÇÃO AMIGA PARA SOBRECARREGAR OS OPERADORES

Como as dicas anteriores indicaram, você pode usar as funções *amigas* para sobrecarregar os operadores dentro de suas classes. No entanto, é importante compreender que existem algumas diferenças entre a sobrecarga normal de operadores e a sobrecarga de operador de função *amiga*. A diferença mais importante é que as funções *amigas* não têm acesso ao ponteiro *this* para a classe. Portanto, seu programa precisa explicitamente passar os operandos para a função do operador *friend* sobrecarregada. Em outras palavras, uma *amiga* que sobrecarrega um operador *unário* recebe um parâmetro, e uma *amiga* que sobrecarrega um operador *binário* recebe dois parâmetros. Como você aprendeu, um operador sobrecarregado na classe recebe um parâmetro a menos do que espera porque ele usa o ponteiro *this*. Quando seus programas sobrecarregam um operador *binário* usando uma função *amiga*, eles precisam passar o operando esquerdo no primeiro parâmetro, e o operando direito no segundo parâmetro. O programa a seguir, *amiga_ms.cpp*, usa uma função *amiga* para sobrecarregar o operador +:

```
#include <iostream.h>

class pos     {
  int longitude, latitude;
  public:
    pos(void) {}   // Usado para construir temporários
    pos (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }

    void exibe(void)
      {
        cout << longitude << " ";
        cout << latitude << endl;
      }

    friend pos operator+(pos op1, pos op2); //  Sobrecarregador amigo
    pos operator=(pos op2);
  };

pos operator+(pos op1, pos op2)
 {
  pos temp;

  temp.longitude = op1.longitude + op2.longitude;
  temp.latitude = op1.latitude + op2.latitude;
  return temp;
 }

pos pos::operator=(pos op2)
 {
  longitude = op2.longitude;
  latitude = op2.latitude;
  return *this;
 }

void main(void)
 {
  pos ob1(10,20), ob2(5,30);

  ob1 = ob1+ob2;
  ob1.exibe();
 }
```

RESTRIÇÕES NA SOBRECARGA DE OPERADOR DE FUNÇÃO AMIGA 953

Exatamente como existem restrições nos operadores de sobrecarga dentro de suas classes, C++ impõe duas restrições na sobrecarga de operadores com as funções *amigas*. Primeiro, você precisa usar um parâmetro de referência para uma classe quando você sobrecarrega o operador de *incremento* ou de *decremento* com uma função *amiga*. A Dica 954, a seguir explica detalhadamente como usar uma função *amiga* para sobrecarregar o operador de *incremento* ou de *decremento*. Segundo, você não pode usar uma função *amiga* para sobrecarregar os operadores listados na Tabela 953.

Tabela 953 Operadores que você não pode sobrecarregar usando uma função *amiga*.

Operadores Restritos	
=	()
[]	->

USANDO UMA FUNÇÃO AMIGA PARA SOBRECARREGAR OS OPERADORES ++ OU -- 954

Se você quiser usar uma função *amiga* para sobrecarregar os operadores de *incremento* ou de *decremento*, precisará passar o operador como um parâmetro de referência. Você precisa passar o parâmetro de referência porque, como aprendeu anteriormente, as funções *amigas* não podem acessar o ponteiro *this*. Além disso, você precisa garantidamente passar o operando como um parâmetro de referência — caso contrário, C++ tratará o operando com um parâmetro *by value*, e não efetuará as operações nos parâmetros que você quiser. Em vez disso, sua função de operador sobrecarregado precisará modificar o operador *by reference* antes de terminar. Para compreender melhor esse processamento, considere o seguinte programa, *amiga_in cpp*, que sobrecarrega os operadores de *incremento* e de *decremento* para a classe *pos:*

```
#include <iostream.h>

class pos
 {
  int longitude, latitude;
  public:
    pos(void) {}    // Usada para construir temporários
    pos (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }
    void exiba(void)
      {
        cout << longitude << " ";
        cout << latitude << endl;
      }
    pos operator=(pos op2);
    friend pos operator++(pos &op1); // Amiga sobrecarregada
    friend pos operator--(pos &op1); // Amiga sobrecarregada
 };

pos pos::operator=(pos op2)
 {
   longitude = op2.longitude;
   latitude = op2.latitude;
   return *this;
```

```
}
pos operator++(pos &op)
{
  op.longitude++;
  op.latitude++;
  return op;
}
pos operator--(pos &op)
{
 op.longitude--;
 op.latitude--;
 return op;
}
void main(void)
{
  pos ob1(10,20), ob2;

  ob1.exibe();
  ++ob1;
  ob1.exibe();          // Exibe 11 e 21
  ob2 = ++ob1;
  ob2.exibe();          // Exibe 12 e 22
  --ob2;
  ob2.exibe();          // Exibe 11 e 21 novamente
}
```

955 RAZÕES PARA SOBRECARREGAR OS OPERADORES COM AS FUNÇÕES AMIGAS

Em muitos casos, o fato de você usar uma função *amiga* ou uma *função-membro* para sobrecarregar um operador não faz diferença funcional no seu programa. Embora sobrecarregar com uma função *amiga* não seja materialmente diferente de usar uma *função-membro* para sobrecartegar, você deverá usar a função-*membro* para alcançar o maior encapsulamento. No entanto, existem algumas situações e uma situação em particular, como você aprenderá nesta dica, na qual uma função *amiga* pode set extremamente útil.

Como você aprendeu, ao usar uma *função-membro* para sobrecarregar um operador binário, o objeto no lado esquerdo do operador gera a chamada para a função do objeto sobrecarregado. Além disso, C++ passa um ponteiro para o objeto no lado esquerdo dentro do ponteiro *this*. Portanto, se você cria uma classe chamada *Cats* e sobrecarrega o operador de adição, o seguinte comando é válido, assumindo que você criou uma ocorrência do objeto chamada *happy*:

```
happy + 100
```

No exemplo anterior, *happy* gera a chamada para a função *de adição* sobrecarregada, que efetua a adição e retorna o valor no ponteiro *this* para *happy*. No entanto, se você escrever a expressão como mostrado no exemplo a seguir, o compilador retornará um erro:

```
100 + happy
```

Como a função de adição sobrecarregada espera receber um objeto de classe que pode referenciar com o ponteiro *this*, a constante que ele recebeu no exemplo anterior fez o compilador retornar um erro. Por outro lado, se você sobrecarregar o operador de adição com um par de funções *amigas*, poderá atingir o mesmo objetivo sem causar um erro. O programa a seguir, *duas_ami.cpp*, usa a classe *pos* para mostrar como usar duas funções *amigas*:

```
#include <iostream.h>

class pos
 {
  int longitude, latitude;
  public:
    pos(void) {}  //  Usada para construir temporários
    pos (int lg, int lt)
      {
         longitude = lg;
         latitude = lt;
      }

    void exibe(void)
      {
         cout << longitude << " ";
         cout << latitude  << endl;
      }

    pos operator=(pos op2);
    friend pos operator+(pos op1, int op2); // Amiga sobrecarregada
    friend pos operator+(int op1, pos op2); // Amiga sobrecarregada
 };

pos pos::operator=(pos op2)
 {
   longitude = op2.longitude;
   latitude = op2.latitude;
   return *this;
 }

pos operator+(pos op1, int op2)
 {
   pos temp;

   temp.longitude = op1.longitude + op2;
   temp.latitude = op1.latitude + op2;
   return temp;
 }
pos operator+(int op1, pos op2)
 {
   pos temp;

   temp.longitude = op1 + op2.longitude;
   temp.latitude = op1 + op2.latitude;
   return temp;
 }

void main(void)
 {
   pos ob1(10,20), ob2(5,30), ob3(7,14);

   ob1.exibe();
   ob2.exibe();
   ob3.exibe();
   ob1 = ob2 + 10;
   ob3 = 10 + ob3;
   ob1.exibe();
```

```
  ob3.exibe();
}
```

956 SOBRECARREGANDO O OPERADOR NEW

Como você aprendeu, seus programas podem sobrecarregar quase qualquer função ou operador. Na verdade, seus programas podem sobrecarregar tanto o operador *new* quanto o *delete*. Você poderia escolher sobrecarregar um desses operadores se quisesse que seus programas usassem algum método especial de alocação. Por exemplo, você poderia querer escrever uma rotina de alocação que viesse a usar o disco rígido para memória virtual se o programa gastasse toda a memória disponível no heap. Qualquer que seja a razão pela qual você queira sobrecarregar a função *new*, o processo é relativamente fácil, como mostrado aqui:

```
#include <stdlib.h>
void *operador new(size_t tamanho)
{
  // Efetua a alocação
  return ponteiro_para_memória;
}
```

O tipo *size_t* precisa ser um tipo capaz de conter a maior memória que a função *new* sobrecarregada pode alocar. O arquivo de cabeçalho *stdlib.h* define o tipo *size_t*. O parâmetro *tamanho* deve conter o número de bytes que *new* requerer para conter o objeto recém-alocado. Finalmente, a função *new* precisa retornar um ponteiro para a memória que aloca ou retornar *NULL* se falhar.

957 SOBRECARREGANDO O OPERADOR DELETE

Exatamente como você pode sobrecarregar o operador *new* para tratar necessidades específicas de alocação de memória, pode sobrecarregar o operador *delete* para liberar a memória que um operador *new* sobrecarregado aloca. O operador *delete* precisa receber um ponteiro para a memória que o operador *new* alocou anteriormente para o objeto. Você pode sobrecarregar ambos os operadores *new* e *delete*, globalmente ou relativo a uma ou mais classes, ou ambos globalmente e em relação a uma ou mais classes. O programa a seguir, *newdel.cpp*, usa as funções *new* e *delete* sobrecarregadas para a classe *pos:*

```
#include <iostream.h>
#include <stdlib.h>

class pos {
  int longitude, latitude;
  public:
    pos(void) {}   //  Usada para construir temporários
    pos (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }
    void show(void)
      {
        cout << longitude << " ";
        cout << latitude << endl;
      }
    void *operator new(size_t tamanho);
    void operator delete(void *p);
};

void *pos::operator new(size_t tamanho)
{
  cout << "Na função new personalizada." << endl;
  return malloc(tamanho);
}
```

```
void pos::operator delete(void *p)
 {
   cout << "Na função delete personalizada." << endl;
   free(p);
 }

void main(void)
 {
   pos *p1, *p2;

   p1 = new pos(10,20);
   if (!p1)
   {
      cout << "Erro de alocação \n";
      exit(1);
   }

   p2 = new pos(-10,-20);
   if (!p2)
   {
      cout << "Erro de alocação\n";
      exit(1);
   }
   p1->show();
   p2->show();
   delete p1;
   delete p2;
   exit (0);
 }
```

Quando você compilar e executar o programa *newdel.cpp*, sua tela exibirá a seguinte saída:

```
Na função new personalizada
Na função new personalizada
10 20
-10 -20
Na função delete personalizada
Na função delete personalizada
C:\>
```

SOBRECARREGANDO NEW E DELETE PARA AS MATRIZES 958

Nas dicas anteriores você aprendeu como sobrecarregar os operadores *new* e *delete* para efetuar alocação de memória personalizada dentro de seus programas. No entanto, se você quer alocar matrizes de objetos, precisa sobrecarregar as funções *new* e *delete* novamente, usando uma forma especial de operador que instrui o compilador de que a sobrecarga é para matrizes. O protótipo para a função *new* sobrecarregada que você pode usar para alocar matrizes dentro de seus programas é mostrada aqui:

```
#include <stdlib.h>

void *operator new[](size_t tamanho)
 {
   // Efetua a alocação
   return ponteiro_para_memoria;
 }
```

Quando você aloca matrizes, C++ automaticamente chama a função construtora da classe para cada objeto na matriz. Quando você libera uma matriz, C++ automaticamente chama a função destrutora do objeto. O programa a seguir, *mat_pers.cpp*, usa as funções sobrecarregadas *new* e *delete* para alocar e liberar o espaço para a matriz:

```
#include <iostream.h>
#include <stdlib.h>

class loc {
  int longitude, latitude;
  public:
    loc(void) {}   //  Usada para construir temporários
    loc (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }

    void exibe(void)
      {
        cout << longitude << " ";
        cout << latitude << endl;
      }

    void *operator new(size_t tamanho);
    void operator delete(void *p);
    void *operator new[](size_t tamanho);
    void operator delete[](void *p);
 };

void *loc::operator new(size_t tamanho)
 {
   cout << "Na função new personalizada." << endl;
   return malloc(tamanho);
 }

void loc::operator delete(void *p)
 {
   cout << "Na função delete personalizada." << endl;
   free(p);
 }

void *loc::operator new[](size_t tamanho)
 {
   cout << "Na função new personalizada alocadora "de MATRIZ" << endl;
   return malloc(tamanho);
 }

void loc::operator delete[](void *p)
 {
   cout << "Liberando a MATRIZ na função delete "personalizada" << endl;
   free(p);
 }

void main(void)
 {
   loc *p1, *p2;
   int i;

   p1 = new loc(10,20);
   if (!p1)
    {
        cout << "Erro de alocação\n";
```

```
        exit(1);
    }

  p2 = new loc[10];
  if (!p2)
   {
     cout << "Erro de alocação\n";
     exit(1);
   }
  p1->exibe();
  for(i=0; i<10; i++)
      p2[i].exibe();
  delete p1;
  delete [] p2;
}
```

O programa *mat_pers.cpp* sobrecarrega os operadores *new* e *delete* para matrizes e objetos individuais. Quando o programa cria uma ocorrência de matriz para um objeto, ele chama o operador *new* personalizado para a matriz e o operador *new* para cada elemento na matriz. O programa efetua processamento similar quando exclui uma ocorrência individual ou uma matriz. Quando você compilar e executar o programa *mat_pers.cpp*, sua tela exibirá a seguinte saída:

```
Na função new personalizada.
Na função new personalizada alocadora de MATRIZ.
10 20
4258096 4258096
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
0 0
Na função delete personalizada.
Liberando a MATRIZ na função delete personalizada.
C:\>
```

SOBRECARREGANDO O OPERADOR DE MATRIZ [] 959

À medida que seus programas forem se tornando mais complexos, algumas vezes você precisará sobrecarregar o operador de *matriz* [] C++ considera o operador de *matriz* como um operador *binário* para propósitos de sobrecarga. Portanto, a forma geral da sobrecarga de uma função de operador de *membro* para o operador de *matriz* [] é:

```
tipo nome_classe::operator[](int i)
 {
   // ...
 }
```

Tecnicamente, o parâmetro *i* no exemplo anterior não precisa ser do tipo *int*; mas, como você tipicamente define matrizes com um parâmetro inteiro, evite usar um parâmetro do tipo *float* ou algum outro tipo. Quando você chamar a função do operador sobrecarregado, C++ atribuirá o ponteiro *this* para o objeto e usará o parâmetro para controlar o tamanho. Para compreender melhor o processamento que a função de matriz [] sobrecarregada efetua, considere o programa a seguir, *mtz_sobr cpp:*

```
#include <iostream.h>

class algumtipo {
  int a[3];
  public:
```

```
    algumtipo(int i, int j, int k)
     {
       a[0] = i;
       a[1] = j;
       a[2] = k;
     }
   int operator[](int i)   { return a[i]; }
};

void main(void)
 {
   algumtipo ob(1, 2, 3);

   cout << ob[1];
 }
```

Sobrecarregar o operador de *matriz* [] lhe dá algum controle importante sobre a criação de matrizes com classes. Além de lhe permitir atribuir valores individuais a membros individuais, você poderá usar a função sobrecarregada para criar um programa que efetue indexação segura de matriz. A indexação segura de matriz lhe ajuda a impedir que seus programas ultrapassem os limites das matrizes durante a execução. O programa a seguir, *mtz_seg.cpp*, estende o programa *mtz_sobr.cpp* para incluir a indexação de matriz segura:

```
#include <iostream.h>
#include <stdlib.h>

class algumtipo {
   int a[3];
  public:
   algumtipo(int i, int j, int k)
    {
      a[0] = i;
      a[1] = j;
      a[2] = k;
    }
   int &operator[](int i);
  };

int &algumtipo::operator[](int i)
 {
   if (i<0 || i>2)
    {
      cout << "Erro de limite.\n";
      exit(1);
    }
   return a[i];
 }

void main(void)
 {
   algumtipo ob(1, 2, 3);

   cout << ob[1];
   cout << endl;
   ob[1] = 25;
   cout << endl;
   cout << ob[1];
   ob[3] = 44;
 }
```

Quando você tentar acessar um objeto fora dos limites da matriz, ele retornará um erro. No caso do programa *mtz_seg.cpp*, tentar acessar o elemento no índice 3 está fora dos limites, de modo que o programa retorna um erro. Quando você executar o programa *mtz_seg.cpp*, ele gerará a seguinte saída:

```
2
25
Erro de limite.
C:\>
```

SOBRECARREGANDO O OPERADOR DE CHAMADA DE FUNÇÃO () 960

Já vimos que C++ lhe permite sobrecarregar muitos de seus operadores dentro de seus programas. Ao sobrecarregar o operador de *chamada de função* (), você não está criando um novo modo de chamar uma função. Em vez disso, você cria uma função *operator* para a qual seus programas podem passar um número arbitrário de parâmetros. Em geral, quando você sobrecarrega o operador de *chamada de função* (), define os parâmetros que quer que seu programa passe para a função sobrecarregada. Para compreender melhor como C++ sobrecarrega o operador de *chamada de função* (), considere o programa a seguir, *sobr_fun.cpp*, que usa o operador de *chamada de função* () sobrecarregado com a classe *pos:*

```
#include <iostream.h>

class pos {
  int longitude, latitude;
  public:
    pos(void) {}    // Usada para construir temporários
    pos (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }
    void exibe(void)
      {
        cout << longitude << " ";
        cout << latitude  << endl;
      }
    pos operator+(pos op2);
    pos operator()(int i, int j);
 };

pos pos::operator()(int i, int j)
 {
  longitude = i;
  latitude = j;
  return *this;
 }

pos pos::operator+(pos op2)
 {
   pos temp;

   temp.longitude = op2.longitude + longitude;
   temp.latitude  = op2.latitude + latitude;
   return temp;
 }

void main(void)
 {
   pos ob1(10,20), ob2(1,1);
```

```
    ob1.exibe();
    ob1(7,8);
    ob1.exibe();
    ob1 = ob2 + ob1(10,10);
    ob1.exibe();
  }
```

No programa *sobr_fun.cpp*, o operador de *chamada de função* () para a classe *pos* lhe permite atribuir novos valores para um objeto que segue com o operador de *chamada de função* (). No programa, o penúltimo comando, *ob1 = ob2 + ob1(10,10)*, utiliza a *chamada de função* () sobrecarregada para atribuir um valor a *ob1* dinamicamente. Neste exemplo em particular, o programa não mantém o valor recém-atribuído. No entanto, se você usasse o operador de *chamada de função* () com um objeto diferente, talvez *ob3*, o programa manteria o valor recém-atribuído dentro desse objeto.

Quando você compilar e executar o programa *fun_sobr.cpp*, sua tela exibirá a seguinte saída:

```
10 20
7 8
11 11
C:\>
```

961 SOBRECARREGANDO O OPERADOR DE PONTEIRO ->

Como você aprendeu, C++ lhe permite sobrecarregar muitos de seus operadores. À medida que seus programas forem se tornando mais complexos, algumas vezes você precisará sobrecarregar o operador de *ponteiro*. Quando você faz isso, precisa primeiro compreender que C++ trata o operador *ponteiro* como um operador unário (isto é, um operador com um único operando) quando você o sobrecarrega Ao sobrecarregar a função *ponteiro*, você precisa retornar um ponteiro para um objeto da classe chamadora. Quando você sobrecarrega o operador *ponteiro ->*, seu valor de retorno é o mesmo que seu programa receberia se ele invocasse o operador ponto com o objeto. Em outras palavras, os comandos a seguir são equivalentes:

```
obj->i = 10;
obj.i = 10;
```

Para compreender melhor como C++ processa os operadores ponteiros -> sobrecarregados, considere o programa *sobr_ptr.cpp*, como mostrado aqui:

```
#include <iostream.h>

class exemplo {
  public:
    int i;
    exemplo *operator->(void)  {return this;}
};

void main(void)
{
  exemplo obj;

  obj->i = 10;   // O mesmo que obj.i
  cout << obj.i << " " << obj->i;
}
```

Nota: Parece não haver qualquer propósito verdadeiramente útil na sobrecarga do operador ->. No entanto, se você vier a descobrir que precisa fazer isso, use a forma mostrada nesta dica.

962 SOBRECARREGANDO O OPERADOR VÍRGULA ,

Seus programas podem sobrecarregar muitos dos operadores de C++. À medida que seus programas e classes forem se tornando mais complexos, você poderá achar que seus programas precisam sobrecarregar o operador *vírgula*. O operador *vírgula* em C++ avalia cada operando em uma lista separada por *vírgula*, e retorna somente o operando mais à direita da lista.

Em outras palavras, se seu código tem uma lista, *E1, E2,* o programa avaliará o operando esquerdo *E1* como uma expressão vazia e retornará sua avaliação de *E2* como o resultado e o tipo da expressão vírgula. Por recursão, o operador vírgula faz seu programa avaliar a expressão *E1, E2, ...,* da esquerda para a direita. O operador vírgula avalia cada *Ei* por sua vez e retorna o valor e tipo de *En* como resultado da expressão inteira. Para evitar ambiguidade entre o operador vírgula e o delimitador vírgula nas listas de argumento de função e nas listas inicializadoras, use parênteses, como mostrado aqui:

```
func(i, (j = 1, j + 4), k);
```

O fragmento de código anterior chama *func* com três argumentos (*i, 5, k*), não quatro Embora seus programas possam sobrecarregar o operador *vírgula* em qualquer modo que você queira, procure manter a consistência com a operação C++ padrão para a vírgula. Para compreender isso melhor, considere o programa a seguir, *sobr_vir.cpp,* que sobrecarrega o operador *vírgula* mas mantém sua operação normal:

```
#include <iostream.h>

class pos {
  int longitude, latitude;
  public:
    pos(void) {}   // Usada para construir temporários
    pos (int lg, int lt)
      {
        longitude = lg;
        latitude = lt;
      }
    void exibe(void)
      {
        cout << longitude << " ";
        cout << latitude  << endl;
      }
    pos operator+(pos op2);
    pos operator,(pos op2);
};

pos pos::operator,(pos op2)
 {
   pos temp;

   temp.longitude = op2.longitude;
   temp.latitude = op2.latitude;
   cout << op2.longitude << "  " << op2.latitude << endl;
   return temp;

pos pos::operator+(pos op2)
 {
   pos temp;

   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitude;
   return temp;
 }

void main(void)
 {
   pos obj1(10,20), obj2( 5,30), obj3(1,1), temp;

   obj1.exibe();
   obj2.exibe();
   obj3.exibe();
   cout << endl;
   obj1 = (obj1, obj2 + obj2, obj2 + obj3);
```

```
    obj1.exibe();  // Exibirá 6,31, obj2 + valor de obj3
}
```

A atribuição da lista operada por vírgula na penúltima linha do programa *sobr_vir.cpp* resulta no comando de atribuição efetuando a atribuição:

```
obj1 = (obj1, obj2 + obj2, obj2 + obj3);
obj1 = obj1;
obj1 = obj2 + obj2;
obj1 = obj2 + obj3;
obj1 = (6, 31);
```

963 COMPREENDENDO A ABSTRAÇÃO

Abstração é o processo de olhar para um objeto em termos de seus métodos (operações), e, ao mesmo tempo, ignorar temporariamente os detalhes subjacentes da implementação do objeto. Os programadores usam abstração para simplificar o projeto e a implementação de programas complexos. Por exemplo, se você quiser escrever um programa processador de texto, a tarefa, a princípio, poderia parecer muito difícil. No entanto, usando abstraçao, você começa a perceber que um processador de texto na verdade consiste de objetos, tais como um objeto documento que você criará, salvará, verificará a ortografia e imprimirá. Visualizando programas em termos abstratos, você poderá compreender melhor a programação requerida. Em C++, a ferramenta principal para suportar a abstração é a classe.

964 ALOCANDO UM PONTEIRO PARA UMA CLASSE

À medida que você for trabalhando com variáveis de classes, algumas vezes desejará alocar matrizes dinâmicas ou listas dinâmicas do tipo da classe. Como você aprendeu, você poderá usar matrizes dinâmicas e listas dinâmicas dentro de seus programas quando você não souber, na hora da compilação, quantos elementos uma matriz ou uma lista irá requerer. Declarar uma matriz dinâmica de objetos de classe é fundamentalmente idêntico a declarar uma matriz dinâmica de qualquer dos tipos básicos de C ou C++. O programa a seguir, *classdin.cpp*, cria uma matriz de ponteiros para variáveis do tipo de classe *Livro:*

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    void exibe_titulo(void) { cout << titulo << '\n'; };
    void exibe livro(void)
    {
      exibe_titulo();
      exibe_editora();
    };
    Livro(char *titulo, char *autor, char *editora, float preco);
  private:
    char titulo[256];
    char autor[64];
    float preco;
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n'; };
 };

Livro::Livro(char *titulo, char *autor, char *editora, float preco)
 {
   strcpy(Livro::titulo, titulo);
   strcpy(Livro::autor, autor);
   strcpy(Livro::editora, editora);
```

```
    Livro::preco = preco;
    cout << "Na construtora." << endl;
  }

void main(void)
  {
    Livro *Biblio[4];
    int i;

    Biblio[0] = new Livro("Bíblia do Programador C/C++",
          "Jamsa e Klander", "Makron Books", 49.95);
    Biblio[1] = new Livro("Bancos de Dados com VB 5",
                          "Ganz", "Makron Books", 54.95);
    Biblio[2] = new Livro("JavaScript para Netscape",
        "Kent e Kent", "Makron Books", 49.95);
    Biblio[3] = new Livro("Aprendendo C++", "Jamsa",
          "Makron Books", 29.95);

    for (i = 0; i < 4; i++)
      Biblio[i]->exibe_livro();
  }
```

Quando você compilar e executar o programa *classdin.cpp*, sua tela exibirá a seguinte saída:

```
Na construtora.
Na construtora.
Na construtora.
Na construtora.
Bíblia do Programador C/C++
Makron Books

Bancos de Dados com VB 5
Makron Books

JavaScript para Netscape
Makron Books

Aprendendo C++
Makron Books
C:\>
```

Toda vez que você cria uma ocorrência usando *new*, C++ chama a função construtora da classe.

DESCARTANDO UM PONTEIRO PARA UMA CLASSE 965

Na dica anterior você criou uma matriz de ponteiros para objetos do tipo *Livro*. Cada vez que o programa criava uma ocorrência, C++ automaticamente chamava a função construtora *Livro*. De um modo similar, se a classe tem uma destrutora, C++ automaticamente chama a função cada vez que o programa destrói uma ocorrência. O programa a seguir, *destrdin.cpp*, soma uma função destrutora para a classe *Livro*. O programa também usa o operador *delete* para descartar o ponteiro para cada ocorrência, como mostrado aqui:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class Livro
 {
  public:
    void exibe_titulo(void) { cout << titulo << '\n'; };
    void exibe_livro(void)
    {
       exibe_titulo();
```

```
        exibe_editora();
    };
    Livro(char *titulo, char *autor, char *editora, float preco);
    ~Livro(void) { cout << "Destruindo a entrada para "
                        << titulo << endl; };
  private:
    char titulo[256];
    char autor[64];
    float preco;
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n'; };
};

Livro::Livro(char *titulo, char *autor, char *editora, float preco)
 {
   strcpy(Livro::titulo, titulo);
   strcpy(Livro::autor, autor);
   strcpy(Livro::editora, editora);
   Livro::preco = preco;
 }

void main(void)
 {
   Livro *Biblio[4];
   int i = 0;

   Biblio[0] = new Livro("Bíblia do Programador C/C++",
       "Jamsa e Klander", "Makron Books", 49.95);
   Biblio[1] = new Livro("Bancos de Dados com VB 5",
                         "Ganz", "Makron Books", 54.95);
   Biblio[2] = new Livro("JavaScript para Netscape",
       "Kent e Kent", "Makron Books", 49.95);
   Biblio[3] = new Livro("Aprendendo C++", "Jamsa",
                         "Makron Books", 29.95);
   for (i = 0; i < 4; i++)
      Biblio[i]->exibe_livro();
   for (i = 0; i < 4; i++)
      delete Biblio[i];
 }
```

Quando você compilar e executar o programa *destrdin.cpp*, sua tela exibirá a seguinte saída:

```
Na construtora.
Na construtora.
Na construtora.
Na construtora.
Bíblia do Programador C/C++
Makron Books

Bancos de Dados com VB 5
Makron Books

JavaScript para Netscape
Makron Books

Aprendendo C++
Makron Books

Destruindo a entrada para Bíblia do Programador C/C++
Destruindo a entrada para Bancos de Dados com VB 5
```

```
Destruindo a entrada para JavaScript para Netscape
Destruindo a entrada para Aprendendo C++
C:\>
```

DESCARTANDO O ESPAÇO EM BRANCO PRELIMINAR NA ENTRADA 966

Como você aprendeu, o canal de E/S *cin* usa espaços em branco como um delimitador para a entrada de dados. Quando você usar *cin*, algumas vezes desejará que ele ignore os espaços em branco preliminares (em outras palavras, o espaço em branco que precede o texto). Nesses casos, seus programas poderão usar o manipulador *ws*, como mostrado aqui:

```
cin >> ws >> buffer;
```

O programa a seguir, *ws.cpp*, usa o manipulador *ws* para remover o espaço em branco preliminar:

```
#include <iostream.h>

void main(void)
 {
   char buffer[256];

   cout << "Digite uma palavra com brancos preliminares"
        << endl;
   cin  >> ws >> buffer;
   cout << "==" << buffer << "==";
 }
```

Para experimentar com o programa *ws.cpp*, remova o manipulador *ws* e altere a entrada de espaço em branco preliminar. Por exemplo, quando você compilar e executar o programa *ws.cpp* e digitar a palavra " Jamsa", sua saída será a mesma como se você tivesse digitado "Jamsa":

```
Digite uma palavra com brancos preliminares
     Jamsa↵
==Jamsa==
C:\>ws
Digite uma palavra com brancos preliminares
Jamsa
==Jamsa==
C:\>
```

COMPREENDENDO AS BIBLIOTECAS DE CLASSES 967

Como você aprendeu, as bibliotecas de objeto tornam a reutilização de funções muito fáceis para seus programas. Uma *biblioteca de classe* é similar a uma biblioteca de classe em que ela contém código que seus programas podem vincular. Ao contrário de uma biblioteca de código-objeto, que contém uma coleção de funções chamáveis, uma biblioteca de classe contém métodos de classe. Para usar os métodos, seus programas precisam usar as estruturas de classes correspondentes. Em outras palavras, seus programas não podem simplesmente chamar as funções da biblioteca de classe sem usar uma classe. Em todo este livro, seus programas utilizaram muito a biblioteca de classe *iostream* de C++ para efetuar operações de E/S usando *cin* e *cout*. Exatamente como você poderá eventualmente criar bibliotecas de código-objeto que contenham as funções que criar, você também poderá criar bibliotecas de classe. Criando suas próprias bibliotecas de classe, você facilitará o uso dos objetos existentes para os futuros programas.

COLOQUE SUAS DEFINIÇÕES DE CLASSE EM UM ARQUIVO DE CABEÇALHO 968

Quando você cria uma classe que outros programas podem usar, deve colocar a declaração da classe em um arquivo de cabeçalho que você baseia no nome da classe. Por exemplo, o arquivo de cabeçalho *iostream.h*, contém

a declaração de classe para a classe *iostream*. Não coloque seus métodos de classe no arquivo de cabeçalho. Em vez disso, compile os métodos de classe e coloque-os em uma biblioteca de classe, como aprendeu anteriormente. Colocar a declaração de classe em um arquivo de cabeçalho torna muito mais fácil para um programa usar uma classe. Em vez de requerer conhecimento da estrutura de classe completa, o programa precisa simplesmente incluir o arquivo de cabeçalho de classe, e, depois, somente usar aqueles membros que ele requer.

969 USANDO A PALAVRA-CHAVE INLINE COM FUNÇÕES-MEMBRO DA CLASSE

Como você sabe, a palavra-chave *in-line* instrui o compilador a colocar o código inline de uma função em cada referência. Usar a palavra-chave *inline* lhe permite aceitar um tamanho maior do programa em troca de um desempenho melhor. Quando você define as funções-membro da classe, C++ lhe permite colocar as funções dentro da própria classe ou fora da classe. Quando você coloca uma definição de função dentro da classe, C++ gera código in-line cada vez que mais tarde encontrar uma invocação do método. No entanto, quando você define a função fora da classe, C++ não usa código in-line. Se você tem um método que definiu fora de uma classe para a qual quer que o compilador gere código in-line, simplesmente preceda o nome da função com a palavra chave *inline*. Por exemplo, a definição de função a seguir instrui o compilador a gerar código in-line para cada invocação do método *exibe_livro*:

```
inline void Livro::exibe_livro(void) {  exibe_titulo();  exibe_editora();  };
```

970 INICIALIZANDO UMA MATRIZ DE CLASSE

Você aprendeu que C++ lhe permite declarar uma matriz de classe. Quando você declarar uma matriz de classe, o programa automaticamente chamará a função construtora para cada elemento. Quando você declara uma matriz de estruturas, C++ lhe permitirá inicializar o membro matriz, como mostrado aqui:

```
struct Funcionario
 {
   char nome[64];
   long id;
 } trabalhadores[2] = {{ "Kris", 1}, {"Happy", 2}};
```

Ao declarar uma matriz de classe, C++ não permitirá que você forneça valores iniciais. Portanto, você poderá achar difícil atribuir os valores-membro. O programa a seguir, *atribmtz.cpp*, usa a função construtora para atribuir cada elemento da matriz:

```
#include <iostream.h>
#include <string.h>

class Funcionario
 {
  public:
    Funcionario(void);
    void exibe_func(void)    { cout << nome << endl; };
  private:
    char nome[256];
    long id;
 };

Funcionario::Funcionario(void)
 {
   static int indice = 0;

   switch (indice++) {
     case 0: strcpy(Funcionario::nome, "Kris");
             Funcionario::id = 1;
             break;
```

```
    case 1: strcpy(Funcionario::nome, "Happy");
            Funcionario::id = 2;
            break;
    };
}

void main(void)
{
   Funcionario trabalhadores[2];

   trabalhadores[0].exibe_func();
   trabalhadores[1].exibe_func();
}
```

A função construtora usa a variável estática *indice* para determinar qual elemento está inicializando. Como você pode imaginar, a construtora pode se tornar muito complicada, dependendo do número de elementos e membros de classe.

DESTRUINDO A MATRIZ DE CLASSE 971

Seus programas podem sobrecarregar o operador *delete* para liberar a memória da matriz de classe para o heap novamente. Você também aprendeu que seus programas mais comumente usarão funções destrutoras para liberar memória ou salvar informações sobre uma classe no disco. Quando você usar um operador *delete* sobrecarregado para liberar matrizes, o operador *delete* chamará a função destrutora para cada elemento dentro da matriz. Quando você escrever o código dentro de sua função destrutora, otimize cuidadosamente o código, o tanto quanto possível para evitar retardo no programa durante as seqüências de destruição. Para compreender melhor o código destrutor especial em que você precisa usar matrizes de classe, considere o programa *mtz_des.cpp*, que soma uma função destrutora ao programa de exemplo que você viu na Dica 958, como mostrado aqui:

```
#include <iostream.h>
#include <stdlib.h>

class pos
{
  int longitude, latitude;
  public:
    pos(void) {}    //  Usada para construir temporários
    ~pos(void);
    pos (int lg, int lt)
     {
        longitude = lg;
        latitude = lt;
     }

    void exibe(void)
      {
        cout << longitude << " ";
        cout << latitude << endl;
      }

    void *operator new(size_t tamanho);
    void operator delete(void *p);
    void *operator new[](size_t tamanho);
    void operator delete[](void *p);
};

pos::~pos(void)
{
  cout << "Na função destrutora" << endl;
}
```

```
void *pos::operator new(size_t tamanho)
 {
   cout << "Na função do operador new personalizada." << endl;
   return malloc(tamanho);
 }

void pos::operator delete(void *p)
 {
   cout << "Na função do operador delete personalizada." << endl;
   free(p);
 }

void *pos::operator new[](size_t tamanho)
 {
  cout << "Na função new de alocação de MATRIZ personalizada" << endl;
  return malloc(tamanho);
 }

void pos::operator delete[](void *p)
 {
   cout << "Liberando a MATRIZ na função delete personalizada" << endl;
   free(p);
 }

void main(void)
 {
   pos *p1, *p2;
   int i;

   p1 = new pos(10,20);
   if (!p1)
    {
       cout << "Erro de alocação\n";
       exit(1);
    }

   p2 = new pos[10];
   if (!p2)
    {
       cout << "Erro de alocação\n";
       exit(1);
    }
   p1->exibe();
   for(i=0; i<10; i++)
       p2[i].exibe();
   delete p1;
   delete [] p2;
 }
```

Quando você executar o programa *mtz_des.cpp*, o programa chamará a destrutora para cada elemento na matriz de classe, depois chamará a destrutora de matriz personalizada e sua tela exibirá a seguinte saída:

```
Na função destrutora
Na função do operador delete personalizada.
Na função destrutora
Na função destrutora
Na função destrutora
Na função destrutora
Na função destrutora
Na função destrutora
Na função destrutora
```

```
Na função destrutora
Na função destrutora
Na função destrutora
Liberando a MATRIZ na função delete personalizada.
```

Quando seu programa chamar a função *delete* personalizada, ele imediatamente disparará a função destrutora para cada objeto na matriz, antes de excluir o ponteiro para a matriz. Portanto, você pode usar as funções destrutoras com matrizes para armazenar valores, gerar outras informações ou remover o valor do objeto.

CRIANDO MATRIZES DE CLASSE INICIALIZADAS 972

Você já sabe que C++ lhe permite criar matrizes de objetos. Ao criar uma matriz de objetos, você pode inicializar os objetos de dois modos: após criar os objetos (usando um laço *for* para percorrer a matriz e inicializar cada elemento dentro da matriz) ou na função construtora do objeto. Por exemplo, o programa a seguir, *simp_ini.cpp*, usa um laço *for* para inicializar uma matriz de variáveis de objetos do tipo *exemplo:*

```
#include <iostream.h>

class exemplo
 {
   int valor;
   public:
     void def_valor(int j)   { valor = j; }
     int le_valor()   { return valor; }
 };

void main(void)
 {
   exemplo obj[3];
   int laco;

   for(laco = 0; laco < 3; laco++)
      obj[laco].def_valor(laco+1);
   for(laco = 0; laco < 3; laco++)
      cout << obj[laco].le_valor() << endl;
 }
```

Quando você compilar e executar o programa *simp_ini.cpp*, sua tela exibirá a seguinte saída:

```
1

3
C:\>
```

Como você pode ver, o programa percorre a matriz em um laço, inicializando cada elemento. Em seguida, ele percorre a matriz novamente, exibindo cada elemento. Embora os laços *while* sejam úteis quando suas matrizes são pequenas, é bom inicializar os valores de sua matriz dentro da declaração da matriz, desse modo simplificando seu programa. O programa a seguir, *ump_ini.cpp*, usa a mesma matriz que o programa *simp_ini.cpp* mostrado anteriormente; porém, *ump_ini.cpp* inicializa a matriz dentro da construtora:

```
#include <iostream.h>

class exemplo
 {
   int valor;
   public:
     exemplo(int j) {valor = j;}      // construtora
     int obtem_valor(void)   {return valor;}
 };

void main(void)
 {
```

```
  int laco;
  exemplo obj[3] = {1, 2, 3};   // inicializadores

  for(laco = 0; laco < 3; laco++)
    cout << obj[laco].obtem_valor() << endl;
}
```

Quando você compilar e executar o programa *ump_ini.cpp*, ele exibirá o mesmo resultado que o programa *simp_ini.cpp*, como mostrado aqui:

```
1
2
3
C:\>
```

Como você pode ver, o programa *ump_ini.cpp* é mais claro e mais curto que o programa *simp_ini.cpp*. Como regra, você deverá inicializar suas matrizes dentro de construtoras se souber quais serão os valores iniciais da matriz. Na Dica 974, você aprenderá como escrever uma classe que suporta matrizes inicializadas e não-inicializadas.

973 INICIALIZANDO UMA MATRIZ COM UMA CONSTRUTORA DE MÚLTIPLOS ARGUMENTOS

Na dica anterior, você aprendeu como inicializar uma matriz de objetos a partir de dentro da função construtora do objeto. O exemplo simples mostrado na Dica 972 inicializou somente um único valor. No entanto, a maioria das classes que você criar conterá múltiplos dados-membro que você poderá querer que seus programas inicializem automaticamente. O programa a seguir, *dosp_ini.cpp*, inicializa uma matriz que contém dois valores dentro da classe:

```
#include <iostream.h>

class exemplo
 {
   int valor1;
   int valor2;
 public:
   exemplo(int j, int k)       // construtora
   {
     valor1 = j;
     valor2 = k;
   }

   int le_valor2()   {return valor2;}
   int le_valor1()   {return valor1;}
 };

void main(void)
 {
   exemplo obj[3] = { exemplo(1,2),
                      exemplo(3,4),
                      exemplo(5,6) };  // inicializadores
   int laco;

   for(laco = 0; laco < 3; laco++)
    {
      cout << "Valor1, Valor 2: ";
      cout << obj[laco].le_valor1();
      cout << ", ";
      cout << obj[laco].le_valor2() << endl;
    }
 }
```

Quando você compilar e executar o programa *doisp_ini.cpp*, sua tela exibirá a seguinte saída:

```
Valor 1, Valor 2: 1, 2
Valor 1, Valor 2: 3, 4
Valor 1, Valor 2: 5, 6
```

CRIANDO MATRIZES INICIALIZADAS VERSUS NÃO-INICIALIZADAS 974

À medida que seus programas ficarem mais complexos, você freqüentemente verá que seus programas precisarão inicializar algumas matrizes de um objeto, mas não necessariamente outras matrizes. Por exemplo, você poderá inicializar duas matrizes que seu programa usará, e deixar não-inicializada uma terceira que seu programa usará somente para o armazenamento temporário. Você poderá sobrecarregar facilmente suas funções construtoras de classe para que suas classes suportem tanto as declarações inicializadas quanto não-inicializadas. O programa a seguir, *inic_uni.cpp*, altera o projeto da classe exemplo para suportar ambos os tipos de declarações:

```
#include <iostream.h>

class exemplo
 {
   int valor;
  public:
   exemplo(void) {valor = 0;}; // construtora não-inicializada
   exemplo(int j)  { valor = j; } construtora inicializada
   int le_valor(void) {return valor;}
 };

void main(void)
 {
   int laco;
   exemplo obj1[3] = {1, 2, 3};
   exemplo obj2[32];

   cout << "Entrando no primeiro laco: " << endl;
   for(laco = 0; laco < 3; laco++)
      cout << obj1[laco].le_valor() << endl;
   cout << "Entrando no segundo laco: " << endl;
   for(laco = 0; laco < 32; laco++)
      cout << obj2[laco].le_valor() << ", ";
   cout << endl;
 }
```

Como você pode ver, o programa *inic_uni.cpp* define tanto a construtora não-inicializada (padrão) quanto uma construtora inicializada. As definições em *main* criam uma matriz de três objetos inicializados e uma matriz de 32 objetos não-inicializados. Quando o programa percorre uma matriz em um laço, a primeira matriz mostra os valores para os quais o programa originalmente inicializou os elementos. No entanto, a segunda matriz mostra 0 para cada elemento, pois a construtora padrão inicializa cada elemento com 0. Quando você compilar e executar o programa *inic_uni.cpp*, sua tela exibirá o seguinte resultado:

```
Entrando no primeiro laço:
1
2
3
Entrando no segundo laço:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

975 TRABALHANDO COM MATRIZES DE CLASSE

Ao trabalhar com matrizes de classe, você tratará as matrizes de classe de forma muito parecida como tratou as matrizes de estrutura quando você trabalhou com elas em C. Como sabe, as matrizes e classes lhe permitem agrupar informações relacionadas. Como você aprendeu, C++ lhe permite criar matrizes de objetos ou usar matrizes como membros de classe. Em geral, C++ não coloca um limite na profundidade na qual seus programas podem ir no número de estruturas de dados embutidas umas nas outras. Por exemplo, a declaração a seguir cria uma matriz de 100 objetos funcionario. Dentro de cada objeto está uma matriz de estrutura *Data* que corresponde à data de contratação do funcionário, a data da primeira avaliação e a da segunda avaliação:

```
class funcionario
 {
   public
     char nome[64];
     int idade;
     char cpf[11];  // Número do CPF
     int categoria;
     float salario;
     unsigned num_func;
     struct Data
     {
        int mes;
        int dia;
        int ano;
     } func_datas[3];
     int tempo_avaliacao(data atual);
 };

// Comandos do programa
void main(void);
 {
   funcionario membros_equipe[100];
   // código do programa
 }
```

Para acessar membros e elementos da matriz, você simplesmente trabalha da esquerda para a direita, iniciando de fora e trabalhando para dentro. Por exemplo, os comandos a seguir atribuem a data de contratação de um funcionário:

```
membros_equipe[10].func_datas[0].mes = 10;
membros_equipe[10].func_datas[0].dia = 31;
membros_equipe[10].func_datas[0].ano = 98;
```

Embora normalmente seja conveniente embutir objetos e matrizes, como esta dica descreve, lembre-se de que quanto mais profundamente seus programas embutirem essas estruturas de dados umas nas outras, mais difíceis de compreender seus dados se tornarão para outros programadores.

Nota: Além das preocupações sobre a clareza das classes embutidas umas nas outras, a definição da classe para a classe funcionario que esta dica define não encapsula bem a classe. Como você aprendeu, seu programa geralmente deve usar funções de interface para manipular e retornar as informações dentro da classe. Na verdade, o encapsulamento eficaz dos dados torna as classes embutidas umas nas outras mais difíceis de usar.

976 COMPREENDENDO COMO AS MATRIZES DE CLASSE USAM A MEMÓRIA

Mesmo com uma grande quantidade de memória disponível para seus programas quando você compra um novo computador, tal como o Pentium, ainda existem limitações em quanta memória seus programas podem acessar. Adicionalmente, quanto maior for um objeto, mais longo será o processamento que ele irá requerer em cada função que manipular esse objeto. Portanto, é importante compreender quanta memória suas classes consumirão.

Você calculará a quantidade de memória que um objeto de uma única classe requer de forma similar à técnica que usou para determinar quanta memória uma única ocorrência de estrutura requer. Calcular a quantidade de memória que uma matriz de classe consumirá é relativamente simples. Primeiro, você precisa determinar a quantidade máxima de espaço que cada ocorrência de classe requer. Para compreender melhor os requisitos de memória da ocorrência de uma classe, considere a seguinte declaração:

```
class exemplo
 {
   public:
     int i, j, k;
     float a, b;
     char c[64];
 }
```

A classe *exemplo* usará 78 bytes de memória para cada ocorrência: 6 bytes para os inteiros, 8 bytes para as variáveis de ponto flutuante e 64 bytes para a matriz de caracteres. Portanto, uma matriz *exemplo* que contenha 10 elementos usará 780 bytes de memória, como mostra a Figura 976.

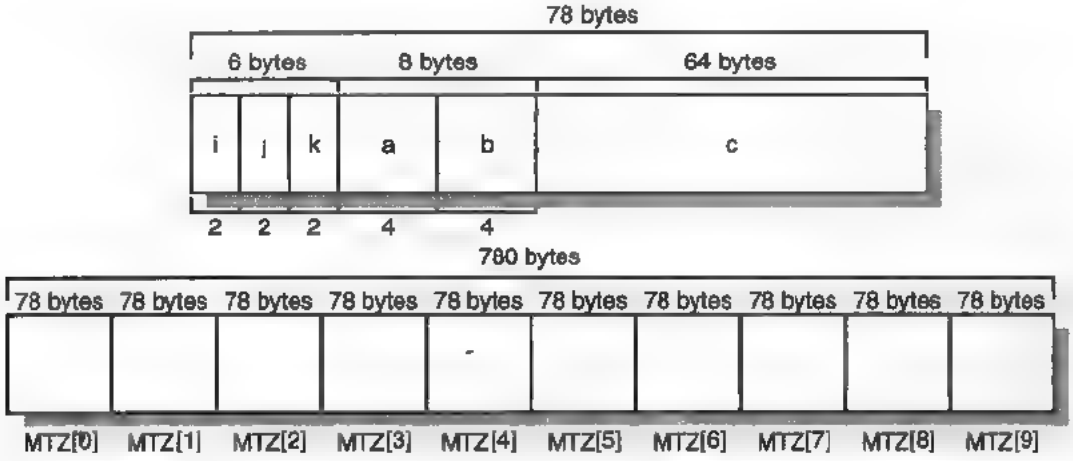

Figura 976 O modelo lógico do consumo de memória da matriz de classe

O Código de Classe in-line Permite Modificações 977

Como você aprendeu, C++ lhe permite colocar funções de método in-line dentro ou fora da classe. Quando você estiver determinando quais funções colocar in-line ou se quaisquer funções deverão ser in-line, lembre-se de que colocar o código do método in-line irá expor o código a modificações. Por exemplo, a classe a seguir usa várias funções in-line:

```
class Livro
 {
  public:
    char titulo[256];
    char autor[64];
    float preco;
    void exibe_titulo(void)  { cout << titulo << '\n'; };
    float le_preco(void)     { return(preco); };
    void exibe_livro(void);
    void atrib_editora(char *nome) {strcpy(editora,nome); };
  private:
    char editora[256];
    void exibe_editora(void) { cout << editora << '\n'; };
 };
```

Quando outro programador usar a classe anterior, ele poderá facilmente modificar os métodos da classe porque o código estará contido dentro da própria classe. Se, em vez disso, você colocar os métodos de classe dentro de uma biblioteca de classe, o programador precisará ter acesso ao código-fonte da biblioteca para alterar os métodos da classe. Usando uma bibl.oteca de classe, você poderá isolar a classe de modificações "não-planejadas".

978 COMPREENDENDO O ARMAZENAMENTO ESTÁTICO

A documentação de C++ se refere ao espaço no heap como a *armazenagem livre*. Lendo artigos e livros sobre C++, você poderá encontrar o termo *armazenagem estática*. No sentido mais simples, o armazenamento estático é uma região de memória global a partir da qual o compilador poderá alocar dados. Quando você criar variáveis globais ou estáticas, o compilador poderá alocar memória para as variáveis a partir da armazenagem estática. Na maioria dos casos, o escopo de objetos que o compilador aloca a partir da armazenagem estática é o programa inteiro. Em outras palavras, os objetos são globais.

979 SINCRONIZANDO AS OPERAÇÕES DE CANAL DE E/S COM STDIO

Como você aprendeu, os programas C++ podem usar as funções de saída padrão, tais como *printf* e *scanf*, que estão definidas em *stdio.h*, ou podem usar os operadores de extração e de inserção com os canais de E/S *cout* e *cin*. Para melhorar a legibilidade do seu programa, normalmente você deve escolher uma técnica ou a outra. No entanto, algumas vezes, você não poderá evitar o uso de ambas. Para esses casos, você pode sincronizar as operações entre *cout* e *cin* usando a função *sync_with_stdio*. Esta função instrui as duas técnicas de E/S a usar o mesmo buffer de entrada e o mesmo buffer de saída para que os mesmos dados sejam acessíveis para ambos. O programa a seguir, *sinc_es.cpp*, ilustra como usar a função *sync_with_stdio*:

```
#include <iostream.h>
#include <stdio.h>

void main(void)
 {
   ios::sync_with_stdio();

   printf("Este livro é ");
   cout << "Bíblia do Programador C/C++, do Jamsa!\n";
 }
```

980 COMPREENDENDO OS CANAIS DE E/S DE C++

Quase todas as dicas apresentadas neste livro usaram muito os canais de E/S cin, cout e cerr. Na seção C++ Avançado, você aprenderá sobre *herança*, o que permite que os objetos de uma classe herdem as características de outra classe. Felizmente, C++ fornece a classe-base *ios* (canal de E/S), que define as operações fundamentais de E/S. Usando o canal *ios*, C++ deriva uma classe de canal de saída e uma classe de canal de entrada. Quando você examinar atentamente o arquivo de cabeçalho *iostream.h* na seção C++ Avançado encontrará uma definição de classe para *ios* e as definições de classe para os canais de entrada e saída de C++, discutidos nas dicas a seguir. Observe que os canais são definidos dentro dos arquivos de cabeçalho *iostream.h*, *fstream.h* e *strstream.h*.

981 COMPREENDENDO OS CANAIS DE SAÍDA DE C++

Em todo este livro, seus programas utilizaram muito o canal de saída *cout*. No sentido mais simples, um *canal de saída* é um destino para os bytes. Na maior parte das discussões anteriores, você pode ter assumido que C++ fornece um canal de saída que *cout*, *cerr* e *clog* usam, e um canal de entrada que *cin* usa. Na verdade, os arquivos de cabeçalho baseados em canais definem três canais de saída. A Tabela 981 descreve rapidamente o uso de cada canal de saída.

Tabela 981 Os canais de saída definidos em *iostream.h*, *fstream.h* e *strstream.h*.

Canal de Saída	Função
ostream	Usado para saída em *cout*, *cerr* e *clog*
ofstream	Usado para gravação de arquivo no disco
ostrstream	Usado para efetuar saída bufferizada de uma string

Várias dicas apresentadas nesta seção discutem modos de seus programas usarem os canais.

Compreendendo os Canais de Entrada de C++ 982

Como você aprendeu na dica anterior, o arquivo de cabeçalho *iostream.h* na verdade define três canais de saída diferentes, um para saída na tela, um para saída de arquivo e um para uso com strings. Como você pode imaginar, os arquivos de cabeçalho *stream* também definem três canais de entrada. Novamente, no sentido mais simples, um *canal de entrada* é uma fonte de bytes. A Tabela 982 descreve rapidamente cada função de canal de entrada.

Tabela 982 Canais de entrada definidos em *iostream.h*, *fstream.h* e *strstream.h*

Canal de Entrada	Função
istream	Usado para entrada a partir de *cin*
ifstream	Usado para leitura de dados de arquivo no disco
istrstream	Usado para leitura da entrada bufferizada de uma string

Várias dicas a seguir discutem os modos de seus programas usarem os canais detalhados nesta dica e na anterior.

Usando os Membros ios para Formatar a Entrada e a Saída 983

Você já sabe que pode usar os membros *ios* para formatar a entrada e a saída. Você também pode usar os tipos enumerados de *sinalizadores de formatação ios*. O arquivo de inclusão *iostream.h* inclui as definições para os tipos enumerados de sinalizadores anônimos. Os valores possíveis para os sinalizadores são mostrados na Tabela 983.

Tabela 983 Os tipos *iosflags* enumerados.

Sinalizador	Valor	Significado
skipws	0x0001	pula o espaço em branco na entrada
left	0x0002	saída ajustada à esquerda
right	0x0004	saída ajustada à direita
internal	0x0008	preenche após o sinal ou indicador de base
dec	0x0010	conversão decimal
oct	0x0020	conversão octal
hex	0x0040	conversão hexadecimal
showbase	0x0080	usa o indicador de base na saída
showpoint	0x0100	força o ponto decimal (saída flutuante)
uppercase	0x0200	saída hexadecimal maiúscula
showpos	0x0400	soma '+' aos inteiros positivos
scientific	0x0800	// usa notação flutuante 1.2345E2
fixed	0x1000	usa notação flutuante 123.45
unitbuf	0x2000	descarrega todos os canais após a inserção
stdio	0x4000	descarrega *stdout*, *stderr* após a inserção
boolalpha	0x8000	insere/extrai bools como texto ou numérico

Por exemplo, se você definir o sinalizador *skipws*, a saída pulará os caracteres de espaço em branco preliminares quando seu programa o processar em um canal. Se você definir o sinalizador *hex*, poderá exibir informações como sua representação hexadecimal.

984 DEFININDO OS SINALIZADORES DE FORMATO

Existem vários modos de usar sinalizadores de formato com um canal. No entanto, o modo mais simples, e de longe o mais comum, é a função-membro *setf*, que seus programas implementarão, como mostrado aqui:

```
#include <iostream.h>

long setf(long flags);
```

Se você chamar a função com um parâmetro vazio, ele retornará as definições anteriores dos sinalizadores de formato. Se você chamar a função com um parâmetro enumerado, ele ligará os sinalizadores que você especificar dentro do parâmetro. Por exemplo, para ligar o sinalizadador *showpos*, você usará a seguinte chamada de função:

```
canal.setf(ios::showpos);
```

O *canal* corresponde ao canal no qual você quer escrever — por exemplo, *cout*, *cerr* e assim por diante. Para compreender melhor esse processamento, considere o programa a seguir, *exib_hex.cpp*, que exibe um valor no formato hexadecimal:

```
#include <iostream.h>

void main(void)
 {
   cout.setf(ios::hex);
   cout.setf(ios::showbase);
   cout << 100;
 }
```

985 LIMPANDO OS SINALIZADORES DE FORMATO

Você pode usar a função *setf* para definir os sinalizadores de formato *ios*. A função *unsetf* limpa os sinalizadores que você definiu anteriormente com *setf*. Como *setf*, *unsetf* é fácil de implementar, como mostrado aqui:

```
#include <iostream.h>

long unsetf(long sinaliz);
```

Por exemplo, você poderia querer definir um sinalizador dentro de um programa que força o computador a escrever tudo em notação científica. No entanto, certas funções no seu programa podem querer desabilitar esse sinalizador antes de gerar sua saída. O programa a seguir, *unsetf.cpp*, liga e depois limpa o sinalizador *uppercase*:

```
#include <iostream.h>

void main(void)
 {
   cout.setf(ios::uppercase | ios::scientific);
   cout << 100.12;
   cout.unsetf(ios::uppercase);
   cout << endl << 100.12;
 }
```

986 USANDO A FUNÇÃO SETF SOBRECARREGADA

Em dicas anteriores, você usou a função *setf* para controlar a saída na tela. Além da implementação de um único parâmetro de *setf*, *iostream.h* fornece uma versão sobrecarregada da função, que você implementará dentro de seus programas, como a seguir:

```
#include <iostream.h>

long setf(long sinaliz1, long sinaliz2);
```

Quando você usar a versão sobrecarregada da função *setf*, seu programa irá zerar os sinalizadores especificados no parâmetro *sinaliz2*, e, depois, definirá esses sinalizadores iguais ao parâmetro *sinaliz1*. Por exemplo, o programa a seguir, *setf_sob.cpp*, limpa os sinalizadores *showpos* e *showpoint* e define o sinalizador *showpoint* novamente:

```
#include <iostream.h>

void main(void)
 {
   cout.setf(ios::showpos | ios::showpoint);
   cout << 100 << endl;
   cout.setf(ios::showpoint, ios::showpos | ios::showpoint);
   cout << 100;
 }
```

Examinando os Sinalizadores de Formatação Atuais 987

À medida que você continuar a trabalhar com os sinalizadores de formatação dentro de seus programas, algumas vezes somente irá querer saber quais são as definições de formato atuais, sem alterar quaisquer definições. Para lhe ajudar a determinar as definições de formato atuais, a classe *ios* fornece a função-membro flags, que retorna a definição atual de cada sinalizador de formato codificada em um inteiro longo. O programa a seguir, *showflag.cpp*, usa uma função personalizada chamada *showflags*, junto com a função *ios* sinalizadores, para gerar informações sobre as definições atuais do sistema:

```
#include <iostream.h>
void showflags(void);

void main(void)
 {
   showflags();
   cout.setf(ios::right | ios::showpoint | ios::fixed);
   showflags();
 }

void showflags(void)
 {
   long sinaliz_ligado, i;
   int j;
   char flags[15][12] = {
      "skipws", "left", "right", "internal", "dec",
      "oct", "hex", "showbase", "showpoint", "uppercase",
      "showpos", "scientific", "fixed", "unitbuf",
   };

   sinaliz_ligado = cout.flags();
   for (i=1, j=0; i<0x2000; i = i<<1, j++)
      if (i & sinaliz_ligado)
         cout << flags[j] << " está ligado."    << endl;
      else
         cout << flags[j] << " está desligado." << endl;
   cout << endl;
 }
```

Quando você compilar e executar o programa *showflag.cpp*, sua tela exibirá a seguinte saída:

```
skipws está ligado.
left está desligado.
right está desligado.
internal está desligado.
dec está desligado.
```

```
oct está desligado.
hex está desligado.
showbase está desligado.
showpoint está desligado.
uppercase está desligado.
showpos está desligado.
scientific está desligado.
fixed está desligado.

skipws está ligado.
left está desligado.
right está ligado.
internal está desligado.
dec está desligado.
oct está desligado.
hex está desligado.
showbase está desligado.
showpoint está ligado.
uppercase está desligado.
showpos está desligado.
scientific está desligado.
fixed está ligado.
C:\>
```

988 DEFININDO TODOS OS SINALIZADORES

Exatamente como a classe *ios* sobrecarrega a função *setf*, ela também sobrecarrega a função *flags*. A função *flags* sobrecarregada lhe permite definir todos os sinalizadores associados a um determinado canal. Em outras palavras, seu programa pode criar uma máscara de sinalizador, chamar a função *flags* e definir todos os sinalizadores detalhados dentro da máscara. Por exemplo, o programa a seguir liga os sinalizadores *showpos*, *showbase*, *oct* e *right* usando a função *flags*. Ele também exibe os sinalizadores antes e após alterar as definições. A função *showflags* é a mesma que a função chamada no programa *showflag.cpp* na dica anterior (que não é reimpresso aqui por questões de espaço, mas é incluído no arquivo do programa *def_tudo.cpp* no CD-ROM que acompanha este livro). No entanto, a função *main* de *def_tudo.cpp* é como mostrado aqui:

```
void main(void)
 {
   showflags();
   long f = ios::showpoint | ios::showbase | ios::oct | ios::right;
   cout.flags(f);
   showflags();
 }
```

989 USANDO A FUNÇÃO PRECISION

Na Dica 830, usamos a função *setprecision* para controlar a precisão com a qual seus programas escrevem dados em ponto flutuante. Você também pode controlar a precisão com a qual *cout* exibe seus dados em ponto flutuante com a função-membro *ios precision*. Você implementará a função-membro *precision* como a seguir:

```
#include <iostream.h>

long precision(int p);
```

Quando seu programa chamar a função *precision*, ele definirá a nova precisão com o número de casas decimais que *p* especifica, e retornará a definição de precisão anterior para a função chamadora. O valor padrão da precisão usado por *cout* são seis casas. Dependendo da versão do seu compilador, você poderá ter que reinicializar a precisão antes de cada chamada a *cout*, ou *cout* usará as definições padrão de precisão. O programa a seguir, *ios_prec.cpp*, usa a função *precision* para formatar a saída:

```
#include <iostream.h>
```

```
void main(void)
 {
   int i;
   float valor = 1.2345;

   for (i = 0; i < 4; i++)
    {
      cout.precision(i);
      cout << valor << endl;
    }
 }
```

Usando a Função fill 990

Como você aprendeu na Dica 828, seus programas podem usar a função-membro *fill* para alterar o caractere que seus canais de saída usam para preencher espaço em branco (o padrão é o caractere de espaço). À medida que seus programas se tornarem mais complexos, usar a função-membro *fill* pode tornar sua saída mais legível e útil para o usuário. Por exemplo, o programa a seguir, *belo_tex.cpp*, usa as funções-membro *width*, *precision* e *fill* para gerar saída formatada:

```
#include <iostream.h>

void main(void)
 {
   cout.precision(4);
   cout.width(10);
   cout << 10.12345 << endl;
   cout.width(10);
   cout.fill('-');
   cout << 10.12345 << endl;
   cout.width(10);
   cout << "Oi!" << endl;
   cout.width(10);
   cout.setf(ios::left);
   cout << 10.12345;
 }
```

Quando você compilar e executar o programa *belo_tex.cpp*, ele gerará o seguinte resultado:

```
     10.12
----10.12
------Oi!
10.12----
C:\>
```

Compreendendo os Manipuladores 991

Nas dicas anteriores, você aprendeu que pode usar comandos dentro do canal de saída *cout* para controlar a exibição de texto. Esses comandos são conhecidos como *manipuladores*. Os manipuladores são úteis porque lhe permitem formatar o texto e, ao mesmo tempo, minimizar o número de comandos usados. Por exemplo, para definir a largura da saída para 10 e o caractere de preenchimento como "*", seus programas podem executar os seguintes comandos:

```
cout.width(10);
cout.fill('*');
cout << "Exemplo" << endl;
```

Alternativamente, seus programas podem usar manipuladores para alcançar o mesmo objetivo, com menos linhas de programação, como mostrado aqui:

```
cout << setw(10) << setfill('*') << "Exemplo" << endl;
```

992 USANDO MANIPULADORES PARA FORMATAR A E/S

Seus programas podem usar manipuladores em vez de funções-membro de canal para formatar a E/S. Os arquivos de cabeçalho *iostream.h* e *iomanip.h* definem os manipuladores. O arquivo de cabeçalho *iomanip.h* define somente manipuladores que recebem parâmetros (tais como *setw*). A Tabela 992 lista os manipuladores que você pode usar dentro de seus programas, o propósito deles, e se você pode usá-los para entrada, saída ou ambas.

Tabela 992 Manipuladores de canais que você pode usar para formatar a E/S.

Manipulador	*Entrada/Saída*	*Propósito*
dec	Entrada/Saída	Instrui o canal a efetuar entrada/saída no formato decimal
endl	Saída	Instrui o canal a escrever um caractere de nova linha e descarregar a si mesmo
ends	Saída	Instrui o canal a escrever um *NULL*
flush	Saída	Instrui o canal a descarregar a si mesmo
hex	Entrada/Saída	Instrui o canal a ler e escrever dados no formato hexadecimal
oct	Entrada/Saída	Instrui o canal a ler e escrever dados no formato octal
resetiosflags(long s)	Entrada/Saída	Desliga os sinalizadores especificados por *s*.
setbase(int base)	Saída	Define a base do número como *base*
setfill(int cr)	Saída	Define o caractere de preenchimento como *cr*.
setiosflags(long s)	Entrada/Saída	Liga os sinalizadores especificados por *s*
setprecision(int p)	Saída	Define o número de dígitos da precisão
setw(int l)	Saída	Define a largura do campo como *l*.
ws	Entrada	Pula o espaço em branco preliminar

993 COMPARANDO OS MANIPULADORES E AS FUNÇÕES-MEMBRO

Como você viu, seus programas podem usar manipuladores, funções-membro ou ambos dentro do código. Por exemplo, o programa a seguir, *usa_man.cpp*, usa um manipulador para controlar a saída no canal:

```
#include <iostream.h>
#include <iomanip.h>

void main(void)
 {
   cout << hex << 100 << endl;
   cout << setfill('?') << setw(10) << 2343.10 << endl;
 }
```

Embora o programa usa_man.cpp possa ter uma compilação ligeiramente mais lenta (pois inclui iomanip.h), seu código será mais compacto que o programa *usa_mem.cpp*, como mostrado aqui:

```
#include <iostream.h>

void main(void)
 {
   cout.setf(ios::hex);
   cout << 100 << endl;
   cout.unsetf(ios::hex);
   cout.fill('?');
   cout.width(10);
   cout << 2343.10 << endl;
 }
```

Quando você compilar os programas *usa_man.cpp* e *usa_mem.cpp*, ambos deverão ser idênticos em tamanho. Ambos os programas terão um tempo de execução idêntico. Em termos de como afetarão seus programas, *ambos os métodos são equivalentes. Você deverá usar o método que achar mais claro, e tornar essa escolha a sua* marca registrada.

Criando Suas Próprias Funções Inserçoras 994

Em muitas dicas anteriores você usou a função inserçora (o operador <<) para colocar saída dentro de um canal. À medida que suas classes se tornarem mais complexas, você poderá querer sobrescrever a função inserçora. Quando você anular a função inserçora, o operador de inserção precisará retornar uma referência a um canal de E/S Todas as funções inserçoras sobrecarregadas que você criar estarão no seguinte formato geral:

```
ostream& operator<<(ostream &canal, tipo_classe objeto)
 {
   // corpo da inserçora
   return canal;
 }
```

A função retorna uma referência do tipo *ostream* — uma classe que C++ deriva da classe *ios* que suporta saída. O primeiro parâmetro para o operador é o canal onde inserir, e o segundo parâmetro é o objeto a enviar para o canal.

Em geral, suas funções inserçoras efetuarão praticamente os mesmos passos que normalmente executam, exceto que farão isso com o objeto canal, em vez de *cout*, *cerr*, ou *clog* de forma direta. Tipicamente, você formatará e estruturará a saída do membro da classe de dados dentro da função de inserção sobrecarregada, como verá na Dica 996, mais à frente.

Sobrecarregando o Operador de Extração 995

Em dicas anteriores, você sobrecarregou vários operadores em relação a classes específicas. Quando suas classes contêm vários membros cujo valor você quer escrever em um modo específico, usar *cout* e o operador de *extração* pode levar a uma quantidade considerável de código. Por exemplo, assuma que você esteja trabalhando com uma classe cujos membros incluem um nome, o sexo (M ou F), a idade e o número telefônico que você quer escrever, como segue:

```
Nome: João Silva     Sexo: M    Idade: 43     Fone: 555-1212
```

Usando *cout*, seus programas precisam incluir o seguinte comando cada vez que quiserem exibir a saída:

```
cout << "Nome: " << nome << "\tSexo: " << sexo
     << "\tIdade: " << idade << "\tFone: " << fone
     << endl;
```

Uma alternativa melhor à forma no programa anterior seria sobrecarregar o operador de extração, como mostrado no programa a seguir, *saisobre.cpp:*

```
#include <iostream.h>
#include <string.h>

class Funcionario
 {
 public:
   Funcionario(char *nome, char sexo, int idade, char *fone)
     {
     strcpy(Funcionario::nome, nome);
     Funcionario::sexo = sexo;
     Funcionario::idade = idade;
     strcpy(Funcionario::fone, fone);
    };
     friend ostream& operator<< (ostream& cout, Funcionario emp);
 private:
   char nome[256];
   char fone[64];
   int idade;
   char sexo;
```

```
};

ostream& operator<< (ostream& cout, Funcionario emp)
 {
   cout << "Nome: " << emp.nome << "\tSexo: "
        << emp.sexo;
   cout << "\tIdade: " << emp.idade << "\tFone: "
        << emp.fone << endl;
   return cout;
 }

void main(void)
 {
   Funcionario trabalhador("Happy", 'M', 4, "555-1212");
   cout << trabalhador ;
 }
```

O programa *saisobre.cpp* sobrecarrega o operador de extração dentro da classe *ostream*. O programa somente usa o operador de extração sobrecarregado quando o programa chama o operador com a classe *Funcionario*. Portanto, C++ não chama a função sobrecarregada quando a sobrecarga efetua operações de saída com a própria sobrecarga, mas trabalha em vez disso com o extrator normal de canal de saída. Como o operador de extração precisa acessar os membros de dados da classe *Funcionario*, o programa declara o operador como um amigo da classe.

996 OUTRO MODO DE SOBRECARREGAR O OPERADOR DE INSERÇÃO DE COUT

A maioria dos programas apresentados nas duas últimas seções deste livro utilizou muito o *iostream cout* para exibir saída na tela. O programa a seguir, *cout_mai.cpp*, sobrecarrega o operador de inserção de *cout* para strings de caracteres, instruindo *cout* a sempre exibir strings de caracteres em maiúsculas:

```
#include <iostream.h>
#include <string.h>

ostream& operator<<(ostream& cout, char *string)
 {
   char *str = strupr(string);

   while (*str)
     cout.put(*str++);
   return(cout);
 }

void main(void)
 {
   cout << "Isto é um teste";
   cout << "\n Bíblia do Programador C/C++, do Jamsa!";
 }
```

Dentro da função de sobrecarga, os comandos usam a função *strupr* para converter a string para maiúscula, e, depois, usam *cout.put* para exibir caracteres um de cada vez. A função de sobrecarga não poderia usar o operador de inserção de *cout* para exibir a string, ou uma série infindável de chamadas recursivas ocorreria.

997 CRIANDO SUAS PRÓPRIAS FUNÇÕES EXTRATORAS

Exatamente como você pode criar suas próprias funções inserçoras, assim também pode sobrecarregar o operador de extração (>>) para as suas classes. Geralmente, você sobrecarregará o operador de extração para obter melhor controle sobre a entrada do usuário nos dados-membro que formam a classe. Sobrecarregue a função extratora usando a seguinte forma geral:

```
istream &operator>>(istream &canal, tipo_classe &objeto)
 {
   // corpo da extratora
   return canal;
 }
```

Observe que, ao contrário das funções inserçoras sobrecarregadas, você precisa passar uma referência ao objeto dentro da função extratora sobrecarregada. As funções extratoras retornam uma referência a um canal do tipo *istream*, que é a classe *stream* de entrada derivada de *ios*. O primeiro parâmetro é uma referência a um canal (geralmente *cin*).

Criando um Exemplo de Extratora 998

Na dica anterior você aprendeu o formato básico para criar uma função extratora sobrecarregada. Embora você possa criar funções globalmente sobrecarregadas, é típico criar uma função extratora sobrecarregada específica para cada classe. Por exemplo, o programa a seguir, *ent_sobr.cpp*, usa a classe *Funcionario* da Dica 995 e recebe entrada projetada especificamente para essa classe:

```
#include <iostream.h>
#include <string.h>

class Funcionario
 {
 public:
   Funcionario(void) {};
   Funcionario(char *nome, char sexo, int idade, char *fone)
   {
     strcpy(Funcionario::nome, nome);
     Funcionario::sexo = sexo;
     Funcionario::idade = idade;
     strcpy(Funcionario::fone, fone);
   };
     friend ostream &operator<<(ostream &cout, Funcionario emp);
     friend istream &operator>>(istream &stream, Funcionario &emp);
 private:
   char nome[256];
   char fone[64];
   int idade;
   char sexo;
 };

ostream &operator<<(ostream &cout, Funcionario emp)
 {
   cout << "Nome: " << emp.nome << "\tSexo: " << emp.sexo;
   cout << "\tIdade: " << emp.idade << "\tFone: " << emp.fone << endl;
   return cout;
 }

istream &operator>>(istream &stream, Funcionario &emp)
 {
   cout << "Digite o nome: ";
   stream >> emp.nome;
   cout << "Informe o sexo: ";
   stream >> emp.sexo;
   cout << "Informe a idade: ";
   stream >> emp.idade;
   cout << "Informe o fone: ";
   stream >> emp.fone;
   return stream;
 }
```

```
void main(void)
 {
   Funcionario trabalhador;

   cin  >> trabalhador;
   cout << trabalhador ;
 }
```

999 CRIANDO SUA PRÓPRIA FUNÇÃO MANIPULADORA

Além de sobrecarregar os operadores de inserção e de extração, seus programas podem criar suas próprias funções manipuladoras. Criar manipuladores personalizados pode ser útil por duas razões. Primeiro, você pode consolidar uma seqüência de várias funções de E/S separadas em uma única chamada de função-membro, expandindo a facilidade de uso e a legibilidade do seu código. Segundo, você pode criar manipuladores personalizados para lhe ajudar a gerenciar a E/S para dispositivos não-padrão. Em geral, você construirá seus manipuladores personalizados da seguinte maneira:

```
nome-canal &nome-manipul(nome-canal &canal [, parâmetros])
 {
   // Código de manipulador específico
   return canal;
 }
```

Nas próximas duas dicas a seguir, você criará algumas funções manipuladoras de saída personalizada. No entanto, você poderá aplicar as técnicas que usará nessas dicas a todas as funções de manipuladores personalizadas.

1000 CRIANDO MANIPULADORES SEM PARÂMETROS

Você aprendeu na dica anterior que pode criar manipuladores personalizados dentro de seus programas. Você criará manipuladores sem parâmetros ou manipuladores parametrizados. A Dica 1001, a seguir, discute os manipuladores parametrizados. No entanto, caso seu manipulador esteja efetuando uma atividade padrão, que não requer entrada adicional a partir do comando chamador, você usará manipuladores *sem parâmetros.* O programa a seguir, *defhexa.cpp,* cria e usa um manipulador sem parâmetro:

```
#include <iostream.h>
#include <iomanip.h>

ostream &sethex(ostream &stream)
 {
   stream.setf(ios::showbase);
   stream.setf(ios::hex);
   return stream;
 }

void main(void)
 {
   cout << 256 << "    " << sethex << 256;
 }
```

1001 USANDO PARÂMETROS COM MANIPULADORES

Na dica anterior, você aprendeu como criar uma função manipuladora sem parâmetros. Você viu que, criar um manipulador é relativamente simples. Infelizmente, criar um manipulador que aceita um ou mais argumentos, tais como *setw (5),* não é simples. Para criar um manipulador que aceita um ou mais parâmetros, você precisa criar o manipulador usando uma *classe genérica.* Como você aprenderá em dicas posteriores, as classes genéricas lhe permitem escrever classes e funções que aceitam múltiplos tipos sem sobrecarregar a classe ou função para cada tipo. Você aprenderá mais sobre as classes genéricas em dicas posteriores e também como escrever uma fun-

ção manipuladora parametrizada na Dica 1124. Por enquanto, considere a seguinte construção de uma típica manipuladora sobrecarregada parametrizada:

```
ostream &recua(ostream &canal, int comprimento)
 {
   register int i;
   for(i = 0; i < comprimento; i++)
      cout << " ";
   return canal;
 }
```

Quando você chamar a manipuladora parametrizada *recua*, o valor que for passado para a manipuladora definirá o número de espaços para o programa recuar dentro do canal.

Compreendendo a Antiga Biblioteca de Classes de Canais 1002

Muitos programas C++ usarão o arquivo de inclusão *iostream.h*, que inclui as definições para muitos dos controles de saída de estilo de C++. Quando Bjarne Stroustrup inventou originalmente C++, a linguagem usava uma biblioteca de classes de E/S menor e ligeiramente diferente, chamada *stream.h*. À medida que C++ continuou a evoluir, ela substituiu a biblioteca *stream.h* pela biblioteca *iostream.h* que este livro descreve. Para manter a compatibilidade retroativa, a maioria dos compiladores C++ continua a suportar a biblioteca *stream.h*. No entanto, você sempre deve usar a mais nova e a mais poderosa biblioteca *iostream.h* dentro dos programas que escrever em C++.

Abrindo um Canal de Arquivo 1003

Como você aprendeu, C++ fornece os canais de arquivo *ifstream* e *ofstream* (entrada e saída). Dentro de seus programas C++, você pode efetuar E/S usando essas duas classes de canais, ou pode usar operações padrão de E/S em arquivo usando *fopen*, *fgets*, *fputs* e assim por diante. Para abrir um canal de arquivo, você precisa declarar uma variável de classe correspondente (*ifstream* ou *ofstream*). Assumindo que você queira efetuar operações de entrada e saída, suas declarações aparecerão como segue.

```
ifstream *entrada;
ofstream saida;
```

Depois, para abrir um canal de arquivo, você poderia usar o membro *open*, como mostrado aqui:

```
entrada.open("NOMEARQ.EXT", ios::in);
saida.open("NOMEARQ.OUT", ios::out);
```

A forma generalizada do comando *open* é mostrada aqui:

```
ifstream.open(const char *NOMEARQ, int nModo=ios::in,
              int nProt=filebuf::openprot);
ofstream.open(const char *NOMEARQ, int nModo=ios::out,
              int nProt=filebuf::openprot);
```

Como você pode ver, o valor padrão para o parâmetro *nModo* é *ios::in* ou *ios::out*, dependendo do canal que você estiver manipulando, de modo que as seguintes construções serão tão válidas como aquelas mostradas anteriormente:

```
entrada.open("NOMEARQ.EXT");
saida.open("NOMEARQ.OUT");
```

O parâmetro *nProt* lhe permite controlar o acesso compartilhado para o arquivo. Além de usar o membro *open*, você pode usar a função construtora do objeto canal quando declarar a variável canal, como mostrado aqui:

```
ifstream entrada ("NOMEARQ.EXT", ios::in);
ofstream saida ("NOMEARQ.OUT", ios::out);
```

Os parâmetros *ios::in* ou *ios::out* selecionam entrada ou saída. Além desses dois valores, seus programas podem usar combinações dos valores listados na Tabela 1003.

Tabela 1003 Valores do modo de abertura para *ifstream* e *ofstream*.

Valor	Significado
ios::app	Abre o canal no modo de inclusão
ios::ate	Abre um arquivo para entrada ou saída, movendo o ponteiro de arquivo para o final do arquivo
ios::in	Abre um arquivo para entrada
ios::out	Abre um arquivo para saída
ios::nocreate	Abre um arquivo somente se ainda ele existir
ios::noreplace	Abre um arquivo somente se ele ainda não existir
ios::trunc	Trunca um arquivo existente
ios.:binary	Abre um arquivo no modo binário

1004 FECHANDO UM CANAL DE ARQUIVO

A dica anterior mostrou que seus programas poderão abrir um *iostream* usando o membro *open* ou usando uma função construtora quando você declarar uma variável stream. Quando você terminar de usar o canal de arquivo, seus programas deverão fechar o stream usando a função-membro *close*, como mostrado aqui:

```
entrada.close();
saida.close();
```

Por padrão, quando seu programa terminar ou se você destruir a variável de classe, C++ fechará o canal de arquivo. No entanto, se você quiser associar o canal de arquivo com um arquivo diferente, precisará primeiro usar o membro *close*, depois usar o novo nome de arquivo para reabrir o canal.

1005 LENDO E GRAVANDO DADOS DO CANAL DE ARQUIVO

Como você aprendeu, C++ permite que seus programas abram canais de arquivo da classe *istream* para operações de entrada. Para ler dados a partir de um canal de arquivo de entrada, seus programas usam o membro *read*, como mostrado aqui:

```
entrada.read(buffer, num_de_bytes);
```

Como você pode ver, o programa precisa fornecer um buffer no qual a função-membro *read* pode armazenar os dados que forem lidos, e o programa precisa informar a função-membro *read* do número de bytes que o programa quer que ele leia. De um modo similar, se você tiver aberto um canal de saída, seus programas podem usar o membro *write* para escrever dados no canal de saída, como a seguir:

```
saida.write(buffer, num_de_bytes);
```

Observe que os membros *read* e *write* não retornam o número de bytes que a função-membro leu ou escreveu com sucesso. Em vez disso, as funções retornam referências ao canal. Para determinar o sucesso ou falha da operação, você precisa verificar os membros de *status* detalhados na dica a seguir. As operações podem falhar por diversas razões, incluindo disco cheio, arquivos não-existentes, erros no disco rígido etc. Você deve ter certeza de que seus programas verificam regularmente o sucesso ou falha da atividade de E/S para se proteger dos erros desconhecidos.

1006 VERIFICANDO O STATUS DE UMA OPERAÇÃO DE ARQUIVO

Quando você verifica as operações de E/S usando os canais de classe *ifstream* e *ofstream*, seus programas podem usar os membros listados na Tabela 1006 para determinar se uma operação de abertura, leitura ou escrita foi bem-sucedida.

Tabela 1006 Funções-membro que retornam informações de sucesso ou falha da E/S.

Membro	Exemplo	Função
bad	*stream.bad()*	Retorna verdadeiro se a operação de E/S encontra um erro irrecuperável
fail	*stream.fail()*	Retorna verdadeiro se a operação de E/S encontra um erro recuperável ou esperado, tal como arquivo não-encontrado

Tabela 1006 Funções-membro que retornam informações de sucesso ou falha da E/S (*Continuação*)

Membro	Exemplo	Função
good	*stream.good()*	Retorna verdadeiro se a operação de E/S é bem-sucedida
eof	*stream.eof()*	Retorna verdadeiro se a operação de E/S encontra um final de arquivo
clear	*stream.clear()*	Limpa os sinalizadores de status
rdstate	*stream.rdstate()*	Retorna o estado de erro atual

Juntando as Operações de Canal de Arquivo 1007

Várias dicas anteriores discutiram canais de arquivo de C++ e as várias funções-membro que seus programas podem usar para efetuar E/S em arquivo. O programa a seguir, *arqcopia.cpp*, usa várias dessas funções-membro para criar um programa simples de cópia de arquivo, que copia arquivos de texto, um caractere de cada vez:

```
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>

void main(int argc, char **argv)
 {
   char buffer[1];
   ifstream entrada(argv[1], ios::in);

   if (entrada.fail())
    {
      cout << "Erro ao abrir o arquivo " << argv[1];
      exit(1);
    }

   ofstream saida(argv[2], ios::out);
   if (saida.fail())
    {
      cout << "Erro ao abrir o arquivo " << argv[2];
      exit(1);
    }
   do {
     entrada.read(buffer, sizeof(buffer));
     if (entrada.good())
       saida.write(buffer, sizeof(buffer));
   } while (! entrada.eof());
   entrada.close();
   saida.close();
 }
```

O programa *arqcopia.cpp* primeiro abre o arquivo que a linha de comando especifica como o arquivo a copiar. Se a ação de abertura de arquivo falhar, o programa retornará com um valor de status de falha. Se o programa abrir com sucesso o primeiro arquivo, ele em seguida abre o arquivo-alvo que sua linha de comando especificou como seu segundo argumento. Se o programa não puder abrir o segundo arquivo, ele novamente encerrará retornando um valor de status de falha. Se o programa abrir com sucesso ambos os arquivos, ele copiará o primeiro arquivo, um caractere de cada vez, para o segundo arquivo. Após o laço que copia caracteres chegar ao final do primeiro arquivo e, portanto, terminar, o programa fechará os canais de arquivos abertos e finalizará normalmente.

Para usar *arqcopia.cpp*, chame o programa como mostrado na listagem a seguir para copiar o arquivo *arqcopia.cpp* para *arqcopia.sav*:

```
C:\>  ARQCOPIA  ARQCOPIA.CPP  ARQCOPIA.SAV   <Enter>
```

1008 EFETUANDO UMA OPERAÇÃO DE CÓPIA BINÁRIA

Na dica anterior você criou o programa *arqcopia.cpp*, que copia o primeiro arquivo de texto especificado na linha de comando para um arquivo com o nome especificado no segundo argumento. Caso queira que o programa copie arquivos binários (tais como um arquivo EXE que contém um programa), você precisará alterar as operações de abertura para usar o sinalizador *ios::binary*, como mostrado aqui:

```
ifstream entrada(argv[1], ios::in | ios:binary);
if (entrada.fail())
 {
   cout << "Erro ao abrir o arquivo " << argv[1];
   exit(1);
 }
ofstream saida(argv[2], ios::out | ios::binary);
if (saida.fail())
 {
   cout << "Erro ao abrir o arquivo " << argv[2];
   exit(1);
 }
```

O CD-ROM que acompanha este livro contém o programa *copiabin.cpp*, que efetua uma operação de cópia binária de arquivo.

1009 COMPREENDENDO A CLASSE STREAMBUF

Nas dicas anteriores você trabalhou com diferentes arquivos e outros canais de E/S para manipular os dados. Quando você bufferiza os dados dentro de seus programas, coloca-os em uma localização interina antes ou após uma operação de leitura ou gravação. C++ deriva as classes que bufferizam dados de E/S (bufferizando canais de E/S) a partir da classe-base *streambuf*. Um canal de bufferização de E/S fornece uma interface de buffer entre seus dados e áreas de armazenagem, tais como memória ou dispositivos físicos. Os buffers que os objetos *streambuf* criam são conhecidos como áreas *get*, *put* e *reserve*. Seus programas acessam e manipulam o conteúdo das áreas criadas pelos objetos *streambuf* com ponteiros que apontam para os caracteres dentro dessas áreas *get*, *put* e *reserve*.

As ações de bufferização que os objetos *streambuf* executam são bem primitivas. Como a bufferização que os objetos *streambuf* efetuam não é tão útil quanto a bufferização que as classes de canais de alto nível que C++ deriva de *streambuf* podem efetuar, seus aplicativos normalmente ganharão acesso aos buffers e funções de bufferização por meio de um ponteiro para *streambuf*. Seus programas usarão esse ponteiro indiretamente, dentro da definição de um objeto baseado em *ios*. A classe *ios* fornece um ponteiro para *streambuf* que fornece acesso transparente para os serviços de buffer para as classes de alto nível — em outras palavras, seus programas devem usar a classe *ios* e deixar o buffer de E/S *streambuf* "atrás dos bastidores" sozinho. As classes de alto nível fornecem formatação de E/S. Em outras palavras, seus programas geralmente devem usar classes de canais de alto nível para controlar suas manipulações de E/S. No entanto, algumas vezes seus programas precisam efetuar acesso de baixo nível em um objeto *streambuf*.

A dica a seguir mostra como acessar os objetos *streambuf* a partir de dentro de seus programas. A Tabela 1009 lista algumas das funções-membro públicas para a classe *streambuf* e suas descrições.

Tabela 1009 Funções-membro da classe streambuf.

Função	Descrição
in_avail	A função-membro in_avail retorna o número de caracteres remanescentes no buffer de entrada interno.
out_waiting	A função-membro out_waiting retorna o número de caracteres remanescentes no buffer de saída interno.
pbump (n)	A função-membro *pbump* incrementa o ponteiro put (*pptr*) por *n*, o que pode ser um valor positivo ou negativo.
sbumpc	A função-membro *sbumpc* retorna o caractere atual a partir do buffer de entrada interno, depois avança o ponteiro interno do buffer para o próximo caractere.

Tabela 1009 Funções-membro da classe *streambuf*. (*Continuação*)

Função	Descrição
sgetc	A função-membro *sgetc* pega o próximo caractere no buffer de entrada interno.
snextc	A função-membro *snextc* avança o ponteiro interno do buffer de entrada para o próximo caractere e retorna esse caractere.
sputbackc	A função-membro *sputbackc* retorna um caractere ao buffer de entrada interno.
sputc	A função-membro *sputc* coloca um caractere no buffer de saída interno.
stossc	A função-membro *stossc* avança o ponteiro interno do buffer de entrada para o próximo caractere no buffer de entrada.

Se você observar as descrições da função-membro cuidadosamente, verá que a maioria das funções-membro para a classe *streambuf* efetua atividades similares às tarefas de alto nível que você usa para *get*, *put*, *read* e *write* executar. Na verdade, como todas as classes baseadas em canais derivam de *streambuf*, todas as funções de nível mais alto que você usa para acessar streams derivam das funções-membro *streambuf*.

Escrevendo um Exemplo Simples com streambuf 1010

Como você aprendeu na dica anterior, C++ fornece a classe-base *streambuf* para lhe ajudar a gerenciar entrada e saída em canais. Embora seus programas devam geralmente usar as classes de nível mais alto e suas funções-membro para controlar os canais, você pode usar uma ocorrência da classe *streambuf* dentro de seus programas de forma muito parecida como usaria uma ocorrência de uma classe de nível mais alto. O programa a seguir, *usa_sbuf.cpp*, usa um objeto *streambuf* para exibir texto digitado do teclado para um arquivo no disco:

```
// Operações com streambufs.
#include <iostream.h>
#include <fstream.h>

void main(void)
 {
   int c;
   const char *nomearq = "_lixo_.$$$";
   ofstream arqsaida;
   streambuf *saida, *entrada = cin.rdbuf();

   // Posiciona no final do arquivo. Anexa todo o texto.
   arqsaida.open (nomearq, ios::ate | ios::app);
   if (!arqsaida)
    {
      cerr << "Não foi possível abrir " << nomearq;
      return (-1);
    }

   saida=arqsaida.rdbuf();            // Conecta ofstream e streambuf

   clog << "Digite algum texto. Use Ctrl+Z para terminar." << endl;
   while ((c = entrada -> sbumpc()) != EOF)
    {
      cout << char(c);    // Ecoa na tela
      if (saida -> sputc(c) == EOF)
         cerr << "Erro na saída";
      }
}
```

Quando você compilar e executar o programa *usa_sbuf.cpp*, ele abrirá o arquivo *_lixo.$$$* no diretório atual. O programa então anexa o objeto *out streambuf* ao buffer para o arquivo *lixo*. O programa examina cada letra que o usuário digita no teclado para ver se não é um Ctrl-Z (*EOF*). O programa usa o membro *streambuf* para exibir cada entrada que não seja *EOF* na tela e para o arquivo. Como você pode ver, esse programa não é

fundamentalmente diferente dos programas que você já escreveu, que usaram as funções-membro *get* e *put* em outras classes baseadas em *stream*.

1011 LENDO DADOS BINÁRIOS USANDO READ

Na Dica 1009, você viu que seus programas podem usar o método *get* da classe *istream* para ler informações um byte de cada vez de um arquivo binário. Seus programas também podem usar a função-membro *read* para ler dados binários a partir de um arquivo um bloco (isto é, algum número de bytes que você determina) de cada vez. Você usará a função *read* dentro de seus programas, como a seguir:

```
#include <iostream.h>
#include <fstream.h>

istream &read(unsigned char *buffer, int num);
```

A função membro *read* lê *num* bytes do canal de entrada. A função-membro *read* então coloca esses *num* bytes no buffer de memória que inicia no endereço para o qual o parâmetro *buffer* aponta. O parâmetro *num* determina o tamanho do bloco que read retorna do arquivo. Em geral, seus blocos devem ser relativamente pequenos, tanto para evitar sobrescrição de memória quanto para manter o arquivo gerenciável no caso de um único *read* falhar. Está claro que, o método *read* é significativamente mais poderoso e mais útil que *get* (porque você pode ler seqüências de dados, em vez de caracteres simples), desde que você tenha algum conhecimento prévio do arquivo que seu programa lerá.

1012 ESCREVENDO DADOS BINÁRIOS USANDO WRITE

Como você aprendeu na Dica 1010, seus programas podem usar a função-membro *put* da classe *ostream* para gravar dados binários em um arquivo. A dica 1011 mostrou que seus programas podem usar o método *read* da classe *istream* para ler um número predeterminado de bytes a partir de um arquivo. Geralmente, você usará o método *read* em conjunção com um arquivo que gravou usando a função-membro *write* da classe *ostream*. Você usará a função-membro *write* dentro de seus programas como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

ostream& write(const unsigned char *buffer, int num);
```

A função-membro *write* grava o número de bytes especificado pelo parâmetro *num* — começando no endereço de memória para o qual *buffer* aponta — para o canal com o qual você chama a função. Para compreender melhor os métodos *read* e *write*, considere o programa a seguir, *lg_estru.cpp*, que grava uma estrutura no disco e depois lê e exibe a estrutura:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>

struct status
 {
   char nome[80];
   float saldo;
   unsigned long numero_conta;
 };

void main(void)
 {
   struct status conta;

   strcpy(conta.nome, "Lars Klander");
   conta.saldo = 1234.56;
   conta.numero_conta = 98765432;
   ofstream saisaldo("saldo.asc", ios::out | ios::binary);
```

```
  if(!saisaldo)
   {
     cout << "Não foi possível abrir o arquivo de saída." << endl;
     exit (1);
   }
  saisaldo.write((unsigned char *) &conta, sizeof(struct status));
  saisaldo.close();
  ifstream inbal("saldo.asc", ios::in | ios::binary);
  if(!inbal)
   {
     cout << "Não foi possível abrir o arquivo." << endl;
     exit (1);
   }
  inbal.read((unsigned char*) &conta,  sizeof(struct status));
  cout << conta.nome << endl;
  cout << "Número da conta: " << conta.numero_conta << endl;
  cout.precision(2);
  cout.setf(ios::fixed);
  cout << "Saldo: $" << conta.saldo << endl;
  inbal.close();
}
```

O programa grava um único registro do tipo *status* no arquivo *saldo.asc*. Como você pode ver, o arquivo *saldo.asc* contém o nome em uma conta, o saldo atual da conta e o numero da conta. O programa grava os valores que o objeto do tipo *status* contém no arquivo *saldo.asc* como uma série de valores binários. O programa então fecha o canal de saída e abre um canal de entrada. O canal de entrada lê os valores binários gravados em uma ocorrência diferente da estrutura *status*. Finalmente, o programa escreve de novo o conteúdo lido do arquivo na tela para o usuário revisar.

USANDO A FUNÇÃO-MEMBRO GCOUNT 1013

Na Dica 1011, você aprendeu que seus programas podem usar o método *read* da classe *istream* para ler um certo número de bytes de um arquivo. No entanto, algumas vezes, uma operação *read* falha, e você deverá determinar quantos caracteres *read* obteve do canal antes de parar. Para fazer isso, seus programas podem usar a função-membro *gcount*, que você usará dentro de seus programas como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

int gcount (void);
```

Quando você chamar a função-membro *gcount* com um canal de entrada, *gcount* retornará o número de caracteres que a última operação de entrada binária leu. Para compreender melhor esse processamento, considere o seguinte programa, *cnt_lido.cpp*, que ilustra como você poderia usar *gcount*:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(void)
{
   float fnum[4] = {99.75, -34.4, 1776.0, 200.1};
   int i;

   ofstream sai("numeros.asc", ios::out | ios::binary);
   if(!sai)
    {
      cout << "Não foi possível abrir o arquivo.";
      exit (1);
    }
```

```
    sai.write((unsigned char *) &fnum, sizeof(fnum));
    sai.close();
    for (i = 0; i < 4; i++)
       fnum[i] = 0.0;
    ifstream in("numeros.asc", ios::in | ios::binary);
    if(!in)
     {
       cout << "Não foi possível abrir o arquivo.";
       exit (1);
     }
    in.read((unsigned char *) &fnum, sizeof(fnum));
    cout << in.gcount() << " bytes lidos." << endl;
    for (i = 0; i < 4; i++)
       cout << fnum[i] << " ";
    in.close();
}
```

O programa *cnt_lido.cpp* define uma matriz de quatro valores em ponto flutuante. Em seguida, o programa grava os quatro valores em um arquivo ASCII. Depois, o programa fecha o canal de saída que ele usou para gravar o arquivo ASCII. Após fechar o canal, o programa limpa os valores na matriz *fnum* e abre um canal de entrada para ler os valores de volta do arquivo. O programa lhe alerta de quantos bytes leu, depois grava os valores em ponto flutuante que leu do disco. Quando você compilar e executar o programa *cnt_lido.cpp*, sua tela exibirá a seguinte saída:

```
16 bytes lidos.
99.75  -34.4  1775  200.1
C:\>
```

1014 USANDO AS FUNÇÕES GET SOBRECARREGADAS

Seus programas podem usar a função-membro *get* da classe *istream* para ler um único caractere de cada vez a partir de um arquivo de texto. No entanto, a maioria dos compiladores também inclui duas versões sobrecarregadas do método *get*. A primeira versão sobrecarregada lê uma série de caracteres; a segunda lê um único *int* de cada vez. Você usará as versões sobrecarregadas do método *get* dentro de seus programas como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

ostream &get(char *buffer, int num, char delimitador='\n');
int get(void)
```

A primeira função sobrecarregada lê caracteres na matriz apontada pelo parâmetro *buffer*. A função lê caracteres até que um entre dois eventos ocorra: ou a função leu o número de caracteres especificado pelo parâmetro *num*, ou encontrou o caractere especificado pelo parâmetro *delimitador*. (Por padrão, o parâmetro *delimitador* é o caractere de nova linha.) Se a função *get* sobrecarregada encontrar o delimitador, irá parar a leitura imediatamente, sem remover o delimitador do canal de entrada.

A segunda função sobrecarregada retorna o próximo caractere do canal como um valor inteiro. Ela retornará a constante *EOF* se encontrar o marcador de arquivo. A segunda função sobrecarregada deve lhe parece familiar, pois é similar à função *getc* de C.

1015 USANDO O MÉTODO GETLINE

Em dicas anteriores, você aprendeu sobre vários métodos que seus programas podem usar para ler informações de um canal de entrada. C++ também fornece o método *getline*, que permite que seus programas leiam dados de um arquivo uma linha de cada vez. Você usará o método *getline* dentro de seus programas, como a seguir:

```
#include <iostream.h>
#include <fstream.h>

ostream &getline(char *buffer, int num, char delimit='\n');
```

Como você pode ver, o método *getline* é quase idêntico ao primeiro método *get* sobrecarregado. O método *getline* lê dados até atingir o delimitador ou até ler o número de caracteres que *num* especifica. O método *getline* coloca os caracteres dentro do buffer que inicia no ponteiro *buffer*. Por exemplo, o programa a seguir, *lelinhas.cpp*, lê o conteúdo de um arquivo de texto uma linha de cada vez e mostra-o na tela:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   if(argc!=2)
    {
      cout << "Uso: PR <nomearq>" << endl;
      exit (1);
    }
   ifstream in(argv[1]);
   if(!in)
    {
      cout << "Não foi possível abrir o arquivo.";
      exit (1);
    }
   char str[255];
   while(in)
    {
      in.getline(str, 255);  // O delimitador por padrão
                             //  é a nova linha
      cout << str << endl;
    }
   in.close();
 }
```

Detectando o Final do Arquivo 1016

Quando você trabalha com canais, seus programas podem determinar quando o ponteiro do arquivo atinge o marcador de final de arquivo comparando o valor que *get* ou uma função correspondente retorna com a constante *EOF*. Seus programas também podem verificar o marcador de final de arquivo verificando o valor do método *eof*, que você usará dentro de seus programas, como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

int eof(void);
```

A função *eof* retornará 0 (falso) normalmente, e retornará um valor diferente de zero (verdadeiro) se o ponteiro estiver no final do arquivo. O programa a seguir, *chec_eof.cpp*, usa *eof* para ler um arquivo a partir de um canal:

```
#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <stdio.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   if(argc!=2)
    {
      cout << "Uso: chec_eof <nomearq>" << endl;
      exit (1);
    }
```

```
   ifstream in(argv[1], ios::in | ios::binary);
   if(!in)
    {
      cout << "Não foi possível abrir o arquivo de entrada." << endl;
      exit (1);
    }

   register int i, j;
   int conta = 0;
   char c[16];

   cout.setf(ios::uppercase);
   while(!in.eof())
    {
      for(i = 0; i < 16 && !in.eof(); i++)
         in.get(c[i]);
      if(i<16)
         i--;                    //não imprime EOF
      for(j = 0; j < i; j++)
         cout << setw(3) << hex << (int) c[j];
      for(; j < 16; j++)
         cout << "   ";
      cout << "\t";
      for(j=0; j < i; j++)
         if(isprint(c[j]))
            cout << c[j];
         else
            cout << ".";
      cout << endl;
      conta ++;
      if(conta==16)
       {
         conta = 0;
         cout << "Pressione ENTER para continuar: ";
         cin.get();
         cout << endl;
       }
    }
   in.close();
}
```

O programa *chec_eof.cpp* lê um arquivo de entrada, 16 bytes de cada vez. Cada vez que o programa lê 16 bytes, ele exibe esses 16 bytes na tela como uma seqüência de números hexadecimais. Em seguida, o programa exibe os mesmos 16 bytes como uma seqüência de letras. Após efetuar o processo de saída 16 vezes, o programa pára e espera que o usuário pressione uma tecla. Depois, ele lê 16 bytes mais 16 vezes (para um total de 256 bytes em cada ciclo) até ler o marcador de final de arquivo. Quando você compilar e executar o programa *chec_eof.cpp* utilizando o arquivo *chec_eof.cpp*, o programa exibirá a seguinte saída, como mostrado aqui parcialmente:

```
23 69 6E 63 6C 75 64 65 20 3C 69 6F 73 74 72 65   #include <iostre
61 6D 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20   am.h>..#include
3C 66 73 74 72 65 61 6D 2E 68 3E  D  A 23 69 6E   <fstream.h>..#in
63 6C 75 64 65 20 3C 63 74 79 70 65 2E 68 3E  D   clude <ctype.h>.
 A 23 69 6E 63 6C 75 64 65 20 3C 69 6F 6D 61 6E   .#include <ioman
69 70 2E 68 3E  D  A 23 69 6E 63 6C 75 64 65 20   ip.h>..#include
3C 73 74 64 69 6F 2E 68 3E  D  A 23 69 6E 63 6C   <stdio.h>..#incl
75 64 65 20 3C 73 74 64 6C 69 62 2E 68 3E  D  A   ude <stdlib.h>..
 D  A 69 6E 74 20 6D 61 69 6E 28 69 6E 74 20 61   ..int main(int a
72 67 63 2C 20 63 68 61 72 20 2A 61 72 67 76 5B   rgc, char *argv[
5D 29  D  A 7B  D  A 20 20 20 69 66 28 61 72 67   ])..{..   if(arg
63 21 3D 32 29  D  A 20 20 20 20 7B  D  A 20 20   c!=2)..    {..
20 20 20 20 63 6F 75 74 20 3C 3C 20 22 55 73 6F       cout << "Uso
3A 20 63 68 65 63 5F 65 6F 66 20 3C 6E 6F 6D 65   : chec_eof <nome
```

```
61 72 71 3E 22 20 3C 3C 20 65 6E 64 6C 3B  D  A   arq>" << endl;..
20 20 20 20 20 20 65 78 69 74 20 28 31 29 3B  D        exit (1);.
Pressione ENTER para continuar:  .
```

USANDO A FUNÇÃO IGNORE 1017

Você aprendeu que C++ fornece muitas ferramentas para ler e gravar nos canais de E/S. No entanto, nenhuma das funções sobre as quais você aprendeu até aqui lhe permitiu ler e descartar caracteres a partir de dentro do canal de entrada. Para efetuar processamento de leitura e descarte, seus programas podem usar a função *ignore*, que você usará dentro de seus programas, como a seguir:

```
#include <iostream.h>
#include <fstream.h>

istream &ignore(int num=1, int delimitador=EOF);
```

A função *ignore* lê e descarta caracteres até que tenha removido *num* caracteres do canal de entrada ou até encontrar o parâmetro *delimitador*, que, por padrão, é o ponteiro *EOF*. Se *ignore* encontrar o *delimitador*, ela não o removerá do canal de entrada. Em vez disso, *ignore* parará seu próprio processamento imediatamente. O programa a seguir, *ignorecr.cpp*, lê a si próprio do disco. O programa *ignorecr.cpp* ignora caracteres até encontrar um espaço ou até que tenha lido 10 caracteres, e, então, o programa exibe o resto do arquivo:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(void)
 {
   ifstream in("ignorecr.cpp");
   if(!in)
    {
      cout << "Não foi possível abrir o arquivo." << endl;
      exit (1);
    }
   in.ignore(10, ' ');
   char c;

   while(in)
    {
     in.get(c);
     cout << c;
    }
   in.close();
 }
```

O código do programa *ignorecr.cpp* usa a função membro *ignore* para pular os primeiros 10 caracteres, ou até o primeiro espaço, o que vier primeiro, dentro do arquivo nomeado. O programa então usa a função *get* para ler os caracteres restantes dentro do arquivo. O programa então usa a função *get* para ler os caracteres restantes dentro do arquivo. O programa exibe na tela cada caractere que ler.

USANDO A FUNÇÃO PEEK 1018

Já vimos que, C++ fornece muitas funções que você pode usar para acessar canais de entrada. À medida que seus programas trabalharem com canais de entrada, algumas vezes você precisará "ver" o próximo caractere dentro de um canal sem remover esse caractere do canal. Para efetuar esse processamento, seus programas podem usar o método *peek*, que você usará dentro de seu programa, como a seguir:

```
#include <iostream.h>
#include <fstream.h>

int peek(void);
```

A função *peek* retornará o próximo caractere no canal ou a constante *EOF* se o ponteiro de arquivo estiver no final do arquivo.

1019 USANDO A FUNÇÃO PUTBACK

Trabalhando com canais, algumas vezes seus programas precisam retornar um caractere que leram no canal de entrada. Você pode usar a função *putback* dentro de seus programas para retornar para um canal de entrada o último caractere que seu programa leu de um determinado canal de entrada. Você usará a função *putback* dentro de seus programas como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

istream &putback(char c);
```

A função *putback* coloca o caractere *c* de volta no canal de onde seu código o chamou e retorna uma referência a esse canal. Observe que *putback* somente retorna um caractere para o canal a partir do qual o programa o recuperou; você não pode usar *putback* para colocar caracteres em outros canais além daqueles a partir dos quais o programa originalmente obteve os caracteres.

1020 ENCONTRANDO A POSIÇÃO ATUAL NO CANAL DE ARQUIVO

Como você aprendeu, há muitos casos em que seus programas querem conhecer a posição atual no arquivo. Na Dica 1016, você usou a função-membro *eof* para determinar se o ponteiro de arquivo estava no final do marcador de arquivo. No entanto, seus programas normalmente precisam saber onde estes estão dentro de um arquivo. Quando seus programas precisam determinar a posição atual do ponteiro de arquivo, podem usar os membros *tellg* e *tellp*, que você usa dentro de seus programas; assim:

```
#include <iostream.h>
#include <fstream.h>

long saida.tellp(void);
long entrada.tellg(void);
```

É importante observar que *tellp* e *tellg* efetuam processamento idêntico. No entanto, seus programas precisam sempre usar *tellp* com canais de saída, e *tellg* com canais de entrada.

1021 CONTROLANDO O PONTEIRO DE CANAL DE ARQUIVO

Os canais de arquivo *ifstream* e *ofstream* fornecem funções-membro que seus programas podem usar para efetuar E/S em arquivo. Quando seus programas efetuam operações de E/S em arquivo, algumas vezes eles precisam definir a posição do ponteiro de canal de arquivo. Para posicionar o ponteiro de arquivo, seus programas podem usar os métodos *seekg* (para entrada) e *seekp* (para saída), como mostrado aqui:

```
#include <iostream.h>
#include <fstream.h>

istream &seekg(streamoff byte_desloc [, seek_dir origem]);
ostream &seekp(streamoff byte_desloc [, seek_dir origem]);
```

O deslocamento de byte é um valor de deslocamento *long* (enumerado dentro do arquivo *iostream.h*) que, a não ser que você especifique de outra forma dentro do parâmetro opcional *origem*, C++ aplica a partir do início do arquivo. Para aplicar o deslocamento de byte a partir de uma posição diferente do início do arquivo, use um dos seguintes valores enumerados para o parâmetro *origem:*

```
enum seek_dir    {  beg=0, cur=1, end=2  };
```

Usando seekg e seekp para Acesso Aleatório 1022

Na dica anterior, você aprendeu sobre os métodos *seekg* e *seekp*, que seus programas podem usar para manipular a localização do ponteiro do arquivo dentro de um arquivo de acesso aleatório. Para compreender melhor como você usará os métodos *seekg* e *seekp* dentro de seus programas, considere o programa a seguir, *muda.cpp*, que usa *seekp* para sobrescrever um determinado caractere dentro de um arquivo:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   if(argc!=4)
    {
      cout << "Uso: muda <nomearq> <byte> <caractere>" << endl;
      exit (1);
    }
   fstream out(argv[1], ios::in | ios::out |  ios::binary);
   if(!out)
    {
      cout << "Não foi possível abrir o arquivo!";
      exit (1);
    }
   out.seekp(atoi(argv[2]), ios::beg);
   out.put(*argv[3]);
   out.close();
 }
```

Quando você compilar e executar o programa *muda.cpp*, ele pesquisará a localização que você especificar dentro do parâmetro *byte* na linha de comando. Ele então substituirá esse byte com o caractere que você especificar no parâmetro *caractere*.

Nota: Se você especificar um valor para byte que está além do ponteiro EOF, o programa gravará um caractere nesse byte e moverá o ponteiro EOF para essa posição. Se você pretender usar um programa que usa processamento similar a muda.cpp para algum propósito maior, certifique-se de colocar um teste no programa que garanta que o valor byte não será maior que o tamanho do arquivo.

Manipulando a Posição do Ponteiro de Arquivo Dentro de um Arquivo 1023

As classes *iostream* de C++ fornecem controle e informações significativas sobre o posicionamento e atividades do ponteiro de arquivo dentro de um arquivo. Ao combinar as informações de acesso aleatório com as informações que os canais podem retornar para seus programas, você terá uma ferramenta poderosa para gerenciar os arquivos de acesso aleatório. O programa a seguir, *seek_tel.cpp*, move-se para várias posições dentro de um arquivo, informa a posição atual dentro do arquivo e exibe o arquivo manipulado quando completar suas atividades. Embora o programa não faça nada de mais, ele torna mais claro o relacionamento entre o ponteiro e os dados do arquivo, como a seguir:

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main(int argc, char *argv[])
 {
   if(argc!=4)
    {
      cout << "Uso: muda <nomearq> <byte> <caractere>" << endl;
      exit(1);
    }
```

```
    fstream out(argv[1], ios::in | ios::out | ios::binary);
    if(!out)
     {
       cout << "Não foi possível abrir o arquivo!";
       exit(1);
     }
    out.seekp(atoi(argv[2]), ios::beg);
    out.put(*argv[3]);
    cout << "A posição atual é: " << out.tellp() << endl;
    out.close();
}
```

O programa *seek_tel.cpp* recebe quatro parâmetros de linha de comando do usuário: o nome do programa, o arquivo no qual deve operar, o número de bytes a deslocar a partir do início do arquivo e um caractere que ele deve colocar dentro do arquivo naquela posição de deslocamento. O programa, então, se move para a posição apropriada dentro do arquivo, exibe o caractere e informa qual é a posição atual. Você deve executar o programa *seek_tel.cpp* com instruções de linha de comando similares às seguintes:

```
C:\> MUDA  TEXTO.TXT  10  A
```

Nota: Exatamente como com a dica anterior, se você especificar um valor para byte que está além do ponteiro EOF, o programa escreverá um caractere nesse byte e moverá o ponteiro EOF para essa posição. Se você pretende usar um programa que usa processamento similar a muda.cpp para algum outro propósito, certifique-se de testar se o valor byte não é maior que o tamanho do arquivo.

1024 DETERMINANDO O STATUS ATUAL DE UM CANAL DE E/S

O sistema de E/S de C++ mantém informações de status sobre o resultado de cada operação de E/S que seu programa executa. C++ contém o estado atual do sistema de E/S dentro de um inteiro cujo valor seus programas podem acessar chamando a função-membro *rdstate*. O inteiro inclui os sinalizadores bit a bit listados na Tabela 1024.

Tabela 1024 Os bits sinalizadores que *ios* enumera.

Sinalizador	Significado
eofbit	1 quando o ponteiro de arquivo atinge o marcador de fim de arquivo; 0 em caso contrário
failbit	1 quando um erro (possivelmente) não-fatal ocorre; 0 em caso contrário
badbit	1 quando um erro fatal de E/S ocorre; 0 em caso contrário

Dentro de seus programas, você pode usar a função-membro *rdstate* para testar o status atual da E/S. Você implementará a função-membro *rdstate* como a seguir:

```
#include <iostream.h>
#include <fstream.h>

int rdstate(void);
```

Para verificar o status atual de um canal de E/S, você usará código similar ao seguinte fragmento:

```
status = in.rdstate();
if (status & ios::eofbit)
   cout << "EOF encontrado." << endl;
if (status & ios::failbit)
   cout << "Erro de E/S não-fatal." << endl;
if (status & ios::badbit)
   cout << "Erro fatal de E/S." << endl;
```

No entanto, exatamente como no tratamento da exceção, seus programas provavelmente procurarão executar um processamento mais útil do que exibir uma mensagem no caso de um erro dessa natureza. Você pode construir um laço *while* que não permite que o usuário saia até que resolva o erro ou selecione Cancelar em uma lista de opções. Exatamente como o tratamento de exceção permite que seus programas efetuem um processa-

mento mais complexo com menos risco de um travamento fatal do sistema, assim também a verificação constante do valor que *rdstate* retorna ajuda a proteger seu programa de E/S que pode levar a uma falha do programa.

Compreendendo as Classes de E/S de Matriz 1025

C++ suporta três classes de E/S com base em matrizes, em contrapartida às classes de E/S com base em arquivos. As três classes com base em matrizes são *istrstream, ostrstream* e *strstream*. C++ deriva todas as três classes a partir de *strstreambuf*, entre outras classes-base. A classe-base *strstreambuf* define vários detalhes de baixo nível que as classes derivadas usam. Além de *strstreambuf*, C++ também deriva *istrstream* a partir de *istream, ostream* e *strstream*.

Devido ao seu lugar dentro da estrutura de herança, todas as três classes stream com base em matrizes têm acesso às mesmas funções-membro que as classes *stream* com base em arquivos. Portanto, você pode usar matrizes stream dentro de seus programas para efetuar muitas das mesmas operações que você efetuaria dentro de um canal de arquivo.

Nota: As classes de E/S com base em matrizes permitem em que seus programas C++ efetuem atividades similares àquelas que seus programas C usavam sprintf para realizar. As classes de E/S com base em matrizes permitem que seus programas bufferizem e formatem a saída antes de exibi-la.

Compreendendo os Canais de String de Caracteres 1026

Na seção deste livro sobre a linguagem C, você aprendeu que seus programas podem usar as funções *sprintf* e *sscanf* para exibir e receber dados de uma string de caracteres. Para lhe ajudar a efetuar operações similares, o arquivo de cabeçalho *ostrstream.h* define a classe *ostrstream*. Quando seus programas criam um canal de saída de string, você essencialmente vincula o canal a uma string de caracteres específica. O programa a seguir, *pree_str.cpp*, cria uma variável do tipo *ostrstream* e a preenche com os caracteres "Bíblia do Programador C/C++, do Jamsa!":

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char string[256];

   ostrstream str(string, 256);   // Vincula a string
   str << "Bíblia do Programador C/C++, do Jamsa!" << ends;
   cout << string;
 }
```

Usando istrstream para Escrever uma String de Caracteres 1027

Na dica anterior vimos como declarar uma variável do tipo *ostrstream* e como usar esse canal de string para exibir informações de dentro de seus programas. As bibliotecas de C++ também definem a classe *istrstream*, que seus programas podem usar para vincular um canal de entrada com uma matriz. Dentro de seus programas, você irá declarar o tipo *istrstream* como mostrado aqui:

```
#include <iostream.h>
#include <strstrea.h>

istrstream istr(char *buffer);
```

Na construtora *istrstream*, o parâmetro *buffer* é um ponteiro para uma matriz que o canal usará como sua fonte para os caracteres. Para lhe ajudar a compreender o processamento que a classe *istrstream* executa, considere o programa a seguir, *prim_ent.cpp:*

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
{
  char ent_string[] = "10 Olá 0x88 12.23 feito";
  istrstream ins(ent_string);
  int i;
  char str[80];
  float f;

  ins >> i;
  ins >> str;
  cout << i << " " << str << endl;
  ins >> i;
  ins >> f;
  ins >> str;
  cout << hex << i << " " << f << " " << str;
}
```

Como os canais de entrada de C++ param de aceitar dados quando encontram espaço em branco, você pode usar o operador de inserção com a string de caracteres *ent_string* para preencher as outras variáveis que o programa define mais tarde ao processar. Quando o programa executa, ele preenche a variável *i* com o valor 10 e a variável *str* com o valor "Olá". O programa então exibe esses valores na primeira linha. Após gerar a primeira linha de saída, o programa novamente preenche *i*, dessa vez com *0x88*. Em seguida, o programa preenche *f* com 12.23, e *str* com "feito". Quando você compilar e executar o programa *prim_ent.cpp*, sua tela exibirá a seguinte saída:

```
10 Olá
88  12.23  feito
C:\>
```

1028 COMPREENDENDO MELHOR OSTRSTREAM

Na Dica 1026, você aprendeu que seus programas podem usar as matrizes de saída *ostrstream* para formatar e projetar a saída antes de você exibi-la. Quando você declarar uma ocorrência de *ostrstream* dentro de seus programas, use a seguinte forma geral:

```
#include <iostream.h>
#include <strstrea.h>

ostrstream ostr(char *buffer, int tamanho, int modo=ios::out);
```

O parâmetro *buffer* contém o endereço inicial da matriz na qual *ostr* escreve a string de saída. O parâmetro *tamanho* contém o tamanho (em bytes) do buffer. Finalmente, o parâmetro opcional *modo* lhe permite controlar como C++ abre o canal. Por padrão, C++ abre o canal para saída normal, mas você pode, por exemplo, alterá-lo para que C++ automaticamente acrescente toda a saída ao canal.

Quando você declarar um canal com base em matriz, seus programas escreverão todas as informações dirigidas ao canal na matriz. No entanto, você deverá ter cuidado para que seus canais não sobrecarreguem o tamanho do buffer, pois seus programas poderão retornar um erro, ou até mesmo provocar um travamento no sistema.

1029 USANDO AS FORMAS ISTRSTREAM SOBRECARREGADAS

A Dica 1027 mostrou que normalmente você declara um canal de entrada com base em matriz com somente um único parâmetro, um ponteiro para a matriz que contém os caracteres de entrada do canal. No entanto, C++ também suporta uma versão sobrecarregada da construtora *istrstream*, cuja forma geral é mostrada aqui:

```
#include <iostream.h>
#include <strstrea.h>

istrstream istr(char *buffer, int tamanho);
```

Você pode usar a construtora sobrecarregada para limitar o acesso do canal a somente os primeiros *tamanho* elementos dentro da matriz para a qual o parâmetro *buffer* aponta. Você poderia usar a construtora sobrecarregada quando conhecesse informações sobre a matriz antecipadamente, ou quando apenas quisesse os primeiros *tamanho* elementos sem se preocupar em descartar o resto. Por exemplo, o programa a seguir, *stre_ent.cpp*, usa a versão sobrecarregada da construtora *istrstream:*

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char ent_string[] = "10 Olá  0x88 12.23 feito";
   istrstream ins(ent_string, 8);
   int i;
   char str[80];
   float f;

   ins >> i;
   ins >> str;
   cout << i << " " << str << endl;
   ins >> i;
   ins >> f;
   ins >> str;
   cout << hex << i << " " << f << " " << str;
 }
```

O programa *stre_ent.cpp* limita o tamanho da matriz que o canal pode acessar aos primeiros 8 bytes. O primeiro *read* trabalha exatamente como no programa *prim_ent.cpp* da Dica 1027, exibindo 10 e "Olá". No entanto, como o canal não pode mais acessar a matriz (pois atingiu o limite de anexação da matriz) após os primeiros 8 bytes, ele começa a ler lixo de algum ponto na memória para dentro do canal. Quando você compilar e executar *str_ent.cpp*, sua tela exibirá a seguinte saída, que claramente não é consistente com o resultado esperado:

```
10 Olá
a 5.9801e-39
C:\>
```

Usando pcount com Matrizes de Saída 1030

Quando seus programas trabalham com matrizes de saída, algumas vezes eles precisam saber quantos caracteres estão na matriz de saída. C++ lhe permite usar a função-membro *pcount* para determinar o número de caracteres que uma string de saída contém. Você implementará a função-membro *pcount* como a seguir:

```
#include <iostream.h>
#include <strstrea.h>

int pcount(void);
```

Se a matriz de saída incluir um terminador *NULL*, *pcount* incluirá o terminador dentro de seu valor retornado (em outras palavras, se a string tiver 15 caracteres de comprimento mais o terminador *NULL*, *pcount* retornará 16). Você pode usar *pcount* com matrizes de saída para controlar melhor o processamento do programa. O programa *pcount.cpp* usa *pcount* para exibir o número de caracteres dentro de um canal de saída, como mostrado aqui:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char str[80];
   ostrstream outs(str, sizeof(str));
```

```
    outs << "Olá ";
    outs << 34 << "   " << 9876.98;
    outs << ends;
    cout << "Tamanho da string: " << outs.pcount() << endl;
    cout << str;
  }
```

O programa *pcount.cpp* acrescenta vários caracteres e valores ao canal *outs*. Após adicionar as informações ao canal, *pcount.cpp* exibe o tamanho, e, depois, o próprio canal. Quando você compilar e executar o programa *pcount.cpp*, sua tela exibirá a seguinte saída:

```
Tamanho da string: 16
Olá 34   9876.98
C:\>
```

1031 MANIPULANDO MATRIZES DE CANAIS COM AS FUNÇÕES-MEMBRO IOS

À medida que seus programas forem trabalhando com canais com base em matrizes, você verá que poderá usar as funções-membro *ios* padrão, tais como *get*, *put*, *rdstate*, *eof* e assim por diante, com seus canais com base em matrizes. Por exemplo, o programa a seguir, *mtz_get.cpp*, lê o conteúdo de uma matriz com o método *get*:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char str[] ="abcdefghijklmnop";
   istrstream ins(str);
   char ch;

   while(!ins.eof())
    {
      ins.get(ch);
      cout << ch << " ";
    }
 }
```

O programa *mtz_get.cpp* anexa o canal *ins* à matriz *str*, depois usa um laço *while* para ler a matriz *str*, um caractere de cada vez e escreve esse caractere na tela. Quando você compilar e executar o programa *mtz_get.cpp*, sua tela exibirá o seguinte:

```
a b c d e f g h i j k l m n o p
C:\>
```

1032 USANDO STRSTREAM

Você aprendeu nas dicas anteriores como criar canais com base em matrizes para entrada e saída. No entanto, algumas vezes seus programas precisam criar um único canal para tratar tanto a entrada quanto a saída. Seus programas podem declarar um único canal que trata E/S total usando a construtora *strstream*. Dentro de seus programas, você irá declarar objetos do tipo *strstream* como mostrado aqui:

```
#include <iostream.h>
#include <strstrea.h>

strstream iostr(char *buffer, int tamanho, int modo);
```

Exatamente como os canais de entrada e saída, *buffer* é um ponteiro para o início da matriz, enquanto *tamanho* representa o número de bytes dentro de *buffer*. Você pode definir o modo usando valores padrão de *ios* enumerados. No entanto, você mais comumente definirá o modo com *ios::in* | *ios::out*. Adicionalmente, será pre-

ciso finalizar suas strings com *NULL*. Para compreender melhor como você implementará canais de E/S com base em matriz, considere o seguinte programa, *matz_es.cpp*, que usa um único canal para efetuar entrada e saída a uma matriz:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
  char iostr[80];
  strstream ios(iostr, sizeof(iostr), ios::in | ios::out);
  char str[80];
  int a, b;

  ios << "10 20 testandotestando";
  ios >> a >> b >> str;
  cout << a << "  " << b << "  " << str << endl;
 }
```

O programa *matz_es.cpp* declara uma matriz de comprimento 80, depois anexa o novo canal *ios* à matriz. Usando a variável *ios*, primeiro o programa lê os valores para dentro do canal, depois exibe os valores da string para as variáveis, e então exibe as variáveis na tela. Quando você compilar e executar o programa *matz_es.cpp*, sua tela exibirá a seguinte saída:

```
10 20 testandotestando
C:\>
```

Efetuando Acesso Aleatório Dentro de uma Matriz Stream

1033

Você pode usar todas as operações padrão de E/S (tais como formatação e geração de strings) em uma matriz *stream*. Portanto, você pode usar os métodos *seekg* e *seekp* para manipular matrizes stream exatamente como faz com os canais com base em arquivo. O programa a seguir, *mtz_alea.cpp*, usa *seekg* para mover o ponteiro de canal dentro de uma matriz stream:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char nome[]="Bíblia do Programador C/C++, do Jamsa!";
   char iostr[80];
   strstream ios(iostr, sizeof(iostr), ios::in | ios::out);
   char cr;

   ios << nome;
   ios.seekg(7, ios::beg);
   ios >> cr;
   cout << "Nome: " << nome << endl;
   cout << "Caractere na posição 8: " << cr;
 }
```

Quando você compilar e executar o programa *mtz_alea.cpp*, sua tela exibirá a seguinte saída:

```
Nome: Bíblia do Programador C/C++, do Jamsa!
Caractere na posição 8: d
C:\>
```

1034 USANDO MANIPULADORES COM MATRIZES STREAM

Como você já aprendeu, C++ lhe permite tratar matrizes stream exatamente como você trataria um canal normal de E/S com base em arquivo. Você usará manipuladores com matrizes stream da mesma forma como os usou anteriormente com os canais de arquivos. O programa a seguir, *mtz_seta.cpp*, usa um par de manipuladores personalizados (*se* para seta à esquerda e *sd* para seta à direita):

```
#include <iostream.h>
#include <strstrea.h>

ostream &sd(ostream &stream)
 {
   stream << "-------->";
   return stream;
 }

ostream &se(ostream &stream)
 {
   stream << "<--------";
   return stream;
 }

void main(void)
 {
   char str[80];
   ostrstream outs(str, sizeof(str));

   outs << sd << "Veja este número: ";
   outs << 100000 << se << ends;
   cout << "  " << str << endl;
 }
```

Quando você compilar e executar o programa *mtz_seta.cpp*, ele exibirá a seguinte saída:

```
-------->Veja este número: 100000<--------
C:\>
```

1035 USANDO UM OPERADOR DE INSERÇÃO PERSONALIZADO COM AS MATRIZES STRING

C++ lhe permite tratar os canais com base em matrizes exatamente como você trataria um canal com base em arquivo. Como as matrizes stream são idênticas de muitos modos aos canais com base em arquivos, você pode criar seus próprios operadores de extração e de inserção da mesma forma como faria com os canais de arquivo. Por exemplo, o programa a seguir, *plota_mz.cpp*, cria uma classe chamada *plota* e usa operadores de inserção personalizados para exibir a saída da classe:

```
#include <iostream.h>
#include <strstrea.h>

const int tamanho = 5;

class plota
 {
   int x, y;
 public:
   plota(int i, int j)
    {
      if(i > tamanho)
        i = tamanho;
```

```
      if(i < 0)
         i = 0;
      if(j > tamanho)
         j = tamanho;
      if(j < 0)
         j=0;
      x = i;
      y = j;
    }
   friend ostream &operator<<(ostream &stream, plota obj);
};

ostream &operator<<(ostream & stream, plota obj)
{
  register int i, j;

  for(j = tamanho; j >= 0; j--)
   {
     stream << j;
     if(j == obj.y)
      {
        for(i = 0; i < obj.x; i++)
           stream << "  ";
           stream << '*';
      }
     stream << endl;
   }
  for(i = 0; i < = tamanho; i++)
     stream << " " << i;
  stream << endl;
  return stream;
}

void main(void)
{
  plota a(2,3), b(1,1);
  char str[200];

  cout << "Saída usando cout:" << endl;
  cout << a << endl << b << endl << endl;
  ostrstream outs(str, sizeof(str));
  outs << a << b << ends;
  cout << "saída usando formatação na RAM:" << endl;
  cout << str;
}
```

No programa *plota_mz.cpp*, o operador de inserção personalizado cria uma grade simples. Depois, ele usa as informações contidas dentro do objeto *plota* para desenhar o único ponto na grade. O programa efetua saída padrão e saída em canal para mostrar que o mesmo operador de inserção trabalha igualmente bem com ambos os canais. Quando você compilar e executar o programa *plota_mz.cpp*, sua tela exibirá a seguinte saída:

```
Saída usando cout:
5
4
3   *
2
1
0
 0 1 2 3 4 5
5
4
```

```
3
2
1    *
0
 0 1 2 3 4 5
saída usando formatação na RAM:
5
4
3    *
2
1
 0 1 2 3 4 5
5
4
3
2
1    *
0
 0 1 2 3 4 5
C:\>
```

1036 USANDO OPERADORES DE EXTRAÇÃO PERSONALIZADOS COM AS MATRIZES STREAM

Como mostrou a dica anterior, você criará operadores de inserção personalizados para suas matrizes de canais exatamente da mesma maneira que faria com operadores de inserção personalizados para um canal com base em arquivo. Criar um operador de extração de matriz stream é similarmente fácil. Você pode criar operadores de extração para suas matrizes stream tão facilmente quanto pode criar operadores de extração para seus canais com base em arquivo. O programa a seguir, *plot_mx2.cpp*, acrescenta um operador de extração personalizado para o programa *plota_mx.cpp* detalhado na dica anterior.

```
#include <iostream.h>
#include <strstrea.h>

const int tamanho = 5;

class plota
  {
   int x, y;
 public:
   plota(void);
   plota(int i, int j)
    {
      if(i > tamanho)
         i = tamanho;
      if(i < 0)
         i = 0;
      if(j > tamanho)
         j = tamanho;
      if(j < 0)
         j=0;
      x = i;
      y = j;
    }
   friend ostream &operator<<(ostream &stream, plota obj);
 };

ostream &operator<<(ostream & stream, plota obj)
 {
   register int i, j;
```

```
  for(j = tamanho; j >= 0; j--)
   {
     stream << j;
     if(j == obj.y)
      {
        for(i = 0; i < obj.x; i++)
           stream << "  ";
           stream << '*';
      }
     stream << endl;
   }
  for(i = 0; i <= tamanho; i++)
     stream << " " << i;
  stream << endl;
  return stream;
}

plota::plota(void)
{
  cout << "Informe o valor x: ";
  cin >> this->x;
  cout << "\nInforme o valor y: ";
  cin >> this->y;
}

void main(void)
{
  plota a(2,3), b(1,1), c;
  char str[200];
  ostrstream outs(str, sizeof(str));

  cout << "Saída usando cout:" << endl;
  cout << a << endl << b << endl << c << endl << endl;
  outs << a << b << c << ends;
  cout << "saída usando formatação na RAM:" << endl;
  cout << str;
  istrstream ins(str);
}
```

A adição do operador de extração personalizado lhe permite colocar na classe os valores que você quer que seus programas desenhem dentro da grade. Ao compilar e executar o programa, ele exibirá ambas as grades que a dica anterior exibe e uma terceira grade que contém seu valor informado de forma personalizada.

Usando Matrizes Dinâmicas com Canais de E/S 1037

Em dicas anteriores você aprendeu como usar a construtora *ostrstream* para criar uma matriz *stream*. Cada vez que você usou a construtora *ostrstream*, declarou o ponteiro inicial e o tamanho da matriz. No entanto, à medida que seus programas se tornarem mais complexos e você ficar mais à vontade com C++, normalmente criará matrizes dinâmicas, em vez de matrizes que tenham um tamanho predefinido. Para criar um canal de saída que usa uma matriz dinâmica, você precisará usar a construtora *ostrstream* de forma ligeiramente diferente dentro de seus programas, como mostrado aqui:

```
#include <iostream.h>
#include <strstrea.h>

ostrstream(void);
```

Quando você usar a construtora sem parâmetro *ostrstream* dentro de seus programas, *ostrstream* criará e manterá uma matriz alocada dinamicamente. Observe que a construtora *ostrstream* não retorna um ponteiro para a matriz. Em vez disso, você precisa usar uma segunda função, *str*, que "congela" a matriz e retorna um ponteiro para ela. Após você congelar uma matriz dinâmica, não poderá usá-la para saída novamente — em vez disso, precisará criar uma nova matriz. O programa a seguir, *din_sai.cpp*, usa uma matriz stream de saída alocada dinamicamente:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char * p;
   ostrstream saida;

   saida << "A Bíblia do ";
   saida << "Programador C/C++ ";
   saida << "tem programas interessantes." << ends;
   p = saida.str();
   cout << p;
   delete p;
 }
```

Quando você compilar e executar o programa, ele colocará as informações que os três operadores de extração contêm no canal *outs*. Em seguida, o programa obtém um ponteiro para o início do canal e exibe o canal usando o ponteiro, em vez da própria variável canal real. Quando você compilar e executar o programa *din_sai.cpp*, sua tela exibirá a seguinte saída:

```
A Bíblia do Programador C/C++ tem programas interessantes.
C:\>
```

1038 COMPREENDENDO OS USOS PARA A FORMATAÇÃO DA MATRIZ STREAM

C++ tem a capacidade de sobrecarregar operadores de extração, manipuladores e quase todas as outras ferramentas de formatação em um canal "vivo" (em outras palavras, com base em arquivo), o que elimina a maior parte da necessidade de formatação com base em RAM (ou seja, matrizes stream). No entanto, existem várias boas razões pelas quais você pode querer usar formatação de matriz stream dentro de seus programas, que esta dica detalha.

Um dos usos mais comuns da formatação com base em matriz é quando seus programas precisam construir uma string que o programa usará mais tarde dentro de uma biblioteca padrão ou uma função de terceiros. Por exemplo, você pode precisar criar uma string que a função de biblioteca padrão *strtok* mais tarde analisará. Outro modo de seus programas poderem usar E/S com base em matriz é quando você cria um editor de texto que efetua operações complexas de formatação — operações de formatação que seriam muito mais demoradas se executadas em um arquivo em vez de em uma matriz.

Finalmente, como o Windows não contém funções padrão que você pode usar para formatar a saída dentro de uma janela, você precisa formatar toda a saída antes de enviá-la para a janela. O uso de matrizes stream no Windows é normalmente muito útil na criação de aplicativos Windows atraentes e úteis.

1039 COMPREENDENDO O MANIPULADOR ENDS

Em dicas anteriores você criou um canal de saída de string e usou o operador de inserção para exibir o manipulador *ends* no canal. O manipulador *ends* coloca um caractere *NULL* no canal, de forma muito parecida à forma como o manipulador *endl* insere o caractere de nova linha. Quando você usar o operador de inserção para inserir texto em um buffer de string, usará o manipulador *ends* regularmente. O programa a seguir, *ends.cpp*, usa o manipulador *ends* com vários canais de saída de string:

```
#include <iostream.h>
#include <strstrea.h>

void main(void)
 {
   char titulo[64], editora[64], autor[64];

   ostrstream titulo_str(titulo, sizeof(titulo));
   ostrstream pub_str(editora, sizeof(editora));
   ostrstream autor_str(autor, sizeof(autor));
```

```
    titulo_str << "Bíblia do Programador C/C++, do Jamsa!" << ends;
    pub_str << "Makron Books" << ends;
    autor_str << "Jamsa e Klander" << ends;
    cout << "Livro: " << titulo << " Editora: " << editora
         << " Autor: " << autor << endl;
}
```

Embora as strings de caracteres normalmente terminem com um valor *NULL*, ao trabalhar com uma matriz stream, você deve explicitamente fornecer o manipulador *ends* para instruir o canal a finalizar o canal. Caso contrário, o canal permanecerá aberto até que seu programa tire um "instantâneo" do canal. O manipulador *ends* não é absolutamente necessário em situações mais simples, tais como aquela mostrada no programa *ends.cpp*, mas pode ser útil em gerenciar mais de perto as matrizes stream para operações de saída mais complexas.

Chamando um Objeto a partir de Outro 1040

À medida que seus programas C++ se tornarem mais complexos, eles começarão a usar mais de um tipo de objeto, e, algumas vezes, um objeto usará outro objeto. Por exemplo, o programa a seguir, *dois_obj.cpp*, cria dois tipos de objetos diferentes: um objeto que contém informações sobre um leitor, e um objeto que contém informações sobre um livro. O objeto *Leitor* chama o objeto *Livro* para exibir informações sobre o livro favorito de um leitor, como mostrado aqui:

```
#include <iostream.h>
#include <string.h>

class Livro  {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo, titulo); };
    void exibe_Livro(void) { cout << titulo; };
  private:
    char titulo[64];
};

class Leitor {
  public:
    Leitor(char *nome)  { strcpy(Leitor::nome, nome); };
    void exibe_Leitor(class Livro Livro)
      {
        cout << "Leitor: " << nome << endl << "Livro: ";
        Livro.exibe_Livro();
      };
  private:
    char nome[64];
};

void main(void)
 {
    Leitor leitor("Kris Jamsa");
    Livro favorito_livro("Fundamentos Sobre Compiladores");

    Leitor.exibe_Leitor(favorito_Livro);
 }
```

O programa *dois_obj.cpp* primeiro cria a ocorrência *leitor* do objeto *Leitor* e acrescenta o nome Kris Jamsa ao membro *nome* privado. Em seguida, o programa cria uma ocorrência do objeto *Livro* chamada *livro_favorito*. Quando o programa chama a função-membro *leitor.exibe_leitor* com o objeto *livro_favorito* como o parâmetro da função-membro, ele chama a função-membro *Livro.Exibe_Livro* para o objeto *livro_favorito*. Quando você compilar e executar o programa *dois_obj.cpp*, sua tela exibirá a seguinte saída:

```
Leitor: Kris Jamsa
Livro: Fundamentos Sobre Compiladores
C:\>
```

1041 FALANDO AO COMPILADOR SOBRE UMA CLASSE

Quando uma classe referencia o identificador para uma segunda classe ainda não declarada dentro do seu programa, você precisa dizer ao compilador que o identificador corresponde a uma classe que você irá declarar posteriormente. Para fazer isso, coloque um comando no seu programa que contenha a palavra-chave *class* e o nome da classe, como a seguir:

```
class nome_classe;
```

Por exemplo, assuma que a classe *Livro* queira dizer ao compilador que a classe *Leitor* é sua amiga. Se você ainda não declarou a classe *Leitor*, poderá colocar o seguinte comando no seu programa que diz ao compilador que você definirá a classe posteriormente no código-fonte:

```
class Leitor;
```

A Dica 1043, à frente, usa essa técnica para informar o compilador sobre a classe *Leitor* antes de referenciar a classe *Leitor* dentro da classe *Livro*.

1042 REVISITANDO AS AMIGAS

Você aprendeu que pode declarar funções *amigas* dentro de seus programas. Frequentemente, você usará funções amigas para sobrecarregar funções-membro e operadores dentro de uma determinada classe. No entanto, como aprendeu, C++ também lhe permite especificar uma classe como amiga de outra, o que permite que a amiga acesse os dados e métodos privados da classe. Embora as funções amigas (que também têm acesso aos métodos e dados privados da classe) normalmente sejam suficientes para a maior parte do processamento, algumas vezes uma classe dentro de seus programas precisa ter acesso complexo aos objetos de outra classe, embora não derivando dessa classe. Nesses casos, você pode usar uma classe *amiga*. As Dicas 1043 e 1044 detalham algumas classes *amigas* de exemplo.

1043 DECLARANDO A CLASSE LEITOR COMO UMA AMIGA

Em dicas anteriores você criou e usou a classe *Livro*, que mantém informações sobre o título, autor, editora e preço de um livro. O programa a seguir usa a palavra-chave *friend* para especificar que a classe *Leitor* é uma amiga da classe *Livro*. Portanto, os objetos do tipo *Leitor* podem acessar os membros privados de um objeto *Livro*. No programa *lelivro.cpp*, a classe *Leitor* acessa o membro de dados privados *titulo* da classe *Livro*, como mostrado aqui:

```
#include <iostream.h>
#include <string.h>

class Leitor;  // Declaração antecipada

class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo, titulo);};
    void exibe_Livro(void) { cout << titulo; };
    friend Leitor;
  private:
    char titulo[64];
 };

class Leitor  {
  public:
    Leitor(char *nome)  { strcpy(Leitor::nome, nome); };
    void exibe_Leitor(class Livro livro)
    {
      cout << "Leitor: " << nome << ' ' << "Livro: " << Livro.titulo;
    };
```

```
  private:
    char nome[64];
};

void main(void)
{
    Leitor leitor("Kris Jamsa");
    Livro Livro_favorito("Fundamentos Sobre Compiladores");

    Leitor.exibe_Leitor(Livro_favorito);
}
```

Como você pode ver dentro do objeto *Livro*, o comando a seguir diz ao compilador que o objeto *Leitor* é um amigo, o que permite que o objeto *Leitor* acesse os membros privados do objeto *Livro:*

```
friend Leitor;
```

OUTRO EXEMPLO DE CLASSE AMIGA 1044

Você já sabe que pode declarar uma classe inteira como *amiga* de outra classe. Conceder à segunda classe o status de *amiga* permite que essa classe acesse todos os membros da primeira classe, incluindo coisas, tais como nomes e constantes enumeradas. O programa a seguir, *troco.cpp*, usa uma *classe* amiga para acessar uma enumeração privada:

```
#include <iostream.h>

class quantia;

class moedas
{
  enum unidades {um, cinco, dez, vintecinco, cinquenta};
  friend quantia;
};

class quantia
{
  moedas::unidades dinheiro;
 public:
   void setm(void);
   int getm(void);
} objeto;

void quantia::setm(void)
{
  dinheiro = moedas::dez;
}

int quantia::getm(void)
{
  return dinheiro;
}

void main(void)
{
  objeto.setm();
  cout << objeto.getm();
}
```

O programa *troco.cpp* declara os tipos de unidades dentro da primeira classe (*moedas*), depois declara a segunda classe (*quantia*) como a que mantém um certo número ou tipo de *unidades*. Como o programa declara a classe *quantidade* como uma amiga da classe *moedas*, a classe *quantia* tem acesso ao membro *unidades* privado.

1045 ELIMINANDO A NECESSIDADE DO COMANDO CLASS NOME_CLASSE

Você aprendeu que, quando uma classe referencia um identificador para uma classe que seu programa ainda não declarou, é possível informar o compilador de que o identificador corresponde a uma classe usando um comando similar ao seguinte:

```
class nome_classe;
```

Incluindo tal comando, a classe pode referenciar uma segunda classe usando somente o identificador da classe, como mostrado aqui:

```
friend nome_classe;
```

Se você colocar *class* entre *friend* e *nome_classe*, eliminará a necessidade da declaração "antecipada":

```
friend class nome_classe;
```

A declaração a seguir da classe *Livro* usa a última técnica para informar o compilador de que a classe *Leitor* é uma amiga:

```
class Livro
 {
  public
    Livro(char *titulo) { strcpy(Livro::titulo, titulo); };
    void exibe_livro(void)  { cout << titulo; };
    friend class Leitor;
  private:
    char titulo[64];
 };
```

1046 RESTRINGINDO O ACESSO A UMA AMIGA

C++ lhe permite especificar que um objeto é um *amigo* de outro, o que dá ao *amigo* acesso aos membros privados do objeto. Para controlar melhor o acesso do *amigo* aos membros privados do objeto, C++ lhe permite especificar os métodos específicos dentro do *amigo* que podem acessar os membros privados. Os outros métodos do amigo não têm acesso aos membros. Por exemplo, assuma que somente o membro *exibe_livro* do objeto *Leitor* requeira acesso aos membros privados do objeto *Livro*. Dentro da classe *Livro*, você pode colocar o seguinte comando:

```
friend Leitor::exibe_livro(void);
```

O programa a seguir, *amigo2.cpp*, usa o formato *friend* restrito para restringir o acesso da classe *Leitor* ao objeto *Livro* para que somente a função *exibe_leitor* possa acessar o objeto *Livro*:

```
#include <iostream.h>
#include <string.h>

class Leitor
 {
  public:
    Leitor(char *nome)  { strcpy(Leitor::nome, nome); };
    void exibe_leitor(class Livro livro);
  private:
    char nome[64];
 };

class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo,titulo); };
```

```
    void exibe_livro(void)    { cout << titulo; };
    friend Leitor::exibe_leitor(Livro livro);
  private:
    char titulo[64];
 };

void Leitor::exibe_leitor(class Livro livro)
  { cout << "Leitor: " << nome << ' ' << "Livro: " << livro.titulo; };

void main(void)
 {
    Leitor Leitor("Kris Jamsa");
    Livro livro_favorito("Fundamentos Sobre Compiladores");

    Leitor.exibe_leitor(livro_favorito);
 }
```

Embora o programa *amigo2.cpp* produza a mesma saída que o programa *dois_obj.cpp*, descrito na Dica 1040, ele é mais cuidadoso, pois dá à classe *Leitor* somente acesso limitado à classe *Livro*. Como você aprendeu, um dos maiores perigos das classes *amigas* é seu acesso ilimitado aos membros internos da classe tomada como *amiga*. Limitar o acesso da classe *amiga* é um passo importante na proteção contra o acesso ilimitado.

Conflito de Nomes e Amigas 1047

Quando suas classes usam *amigas* para acessar os membros de outra classe, algumas vezes os nomes de membros nas duas classes entram em conflito. Quando isso ocorre, o programa usa o membro da classe atual. O programa a seguir, *memconfl.cpp*, ilustra um conflito de nome de membro entre *amigas*:

```
#include <iostream.h>
#include <string.h>

class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo,titulo); };
    void exibe_livro(void)    { cout << titulo; };
    friend class Leitor;
  private:
    char titulo[64];
 };

class Leitor
 {
  public:
    Leitor(char *nome) { strcpy(Leitor::nome, nome); };
    void exibe_leitor(class Livro livro) {
          cout << "Leitor: " << nome << ' ' << "Livro: " << livro.titulo; };
    void exibe_livro(void) { cout << "O leitor do livro é " << nome <<
     endl; } ;
  private:
    char nome[64];
 };

void main(void)
 {
    Leitor leitor("Kris Jamsa");
    Livro livro_favorito("Fundamentos Sobre Compiladores");
    leitor.exibe_livro();
    leitor.exibe_leitor(livro_favorito);
 }
```

Como você pode ver, ambas as classes usam o nome de membro *exibe_livro*. Quando você compilar e executar o programa *memconfl.cpp*, ele usará o membro da classe *Leitor*, como mostrado aqui:

```
O leitor do livro é Kris Jamsa
Leitor: Livro do Kris Jamsa  Livro: Fundamentos Sobre Compiladores
C:\>
```

1048 HERANÇA EM C++

Em dicas anteriores, você aprendeu brevemente que C++ suporta *herança*, o que lhe permite derivar uma nova classe a partir de uma classe-base ou existente. Quando você derivar uma classe a partir de outra em C++, usará o seguinte formato:

```
class classe_derivada:  public classe_base  {
   public:
     // Membros públicos da classe derivada
   private:
     // Membros privados da classe derivada
};
```

Como um exemplo das classes derivadas, o programa a seguir, *cria_bib.cpp*, cria uma classe-base chamada *Livro*, e, depois, deriva uma classe chamada *FichaBiblioteca* a partir da classe-base *Livro*:

```
#include <iostream.h>
#include <string.h>

class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo, titulo); };
    void exibe_titulo(void) { cout << titulo << endl; };
  private:
    char titulo[64];
 };

class FichaBiblioteca : public Livro
 {
  public:
    FichaBiblioteca(char *titulo, char *autor, char *editora) : Livro(titulo)
      {
         strcpy(FichaBiblioteca::autor, autor);
         strcpy(FichaBiblioteca::editora, editora);
      };
    void exibe_biblio(void)
 {
        exibe_titulo();
        cout << autor << ' ' << editora;
    };
  private:
    char autor[64];
    char editora[64];
 };

void main(void)
 {
   FichaBiblioteca ficha("Bíblia do Programador C/C++", "Jamsa e Klander",
     "Makron Books");
   ficha.exibe_biblio();
 }
```

Herança é criticamente importante na programação orientada a objetos. A capacidade de criar uma hierarquia de classe, da mais geral para a mais específica, permite que você controle melhor seus programas. Além

disso, ela facilita a compreensão de seus programas para os outros leitores, e torna os programas mais amplos. Embora as classes sejam uma ferramenta de programação valiosa, herança e suas capacidades associadas são o que realmente liberam o poder da programação C++. Você aprenderá mais sobre a herança nas dicas a seguir.

COMPREENDENDO AS CLASSES BASE E DERIVADA 1049

Um fundamento do conceito de herança é o relacionamento entre classes-base e classes derivadas. Quando você cria uma classe usando uma classe existente, a nova classe herda as características da classe existente. As características da classe existente incluem dados e métodos, bem como seus acessos (públicos e privados). Lendo revistas e livros sobre programação orientada a objetos, você encontrará os termos *classe-base* e *classe derivada*. A classe-base é a classe original cujas características outras classes herdam. A classe derivada é a classe que seu programa cria a partir da classe-base. Muitas classes diferentes podem usar uma classe-base. Da mesma forma, você pode criar uma classe derivada a partir de várias classes-base diferentes. A Figura 1049 mostra uma simples derivação a partir da classe-base.

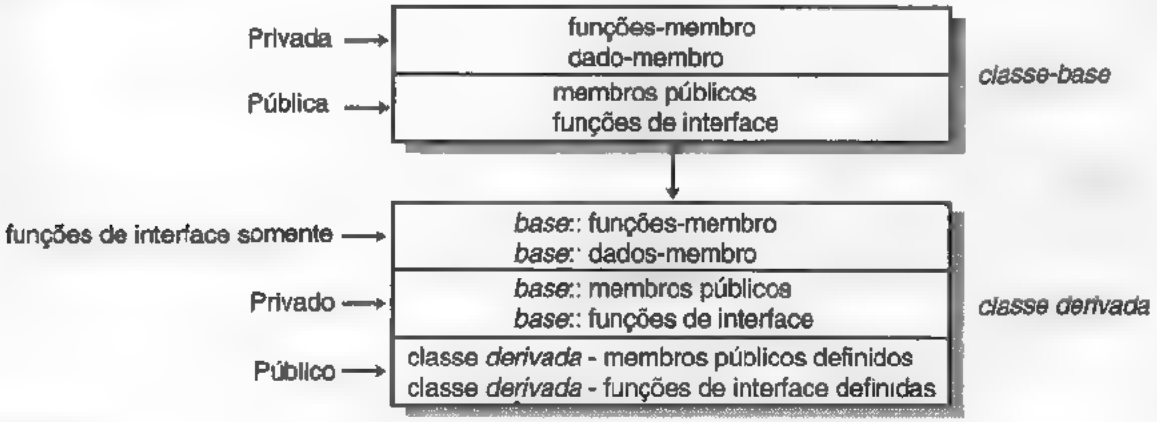

Figura 1049 Um exemplo simples de herança.

DERIVANDO UMA CLASSE 1050

Na Dica 1048, o programa *cria_bib.cpp* derivou a classe *FichaBiblioteca* usando a classe-base *Livro*. A primeira linha da declaração de classe *FichaBiblioteca* informa o compilador de que *FichaBiblioteca* é uma classe derivada que usa a classe-base *Livro*, como mostrado aqui:

```
class FichaBiblioteca : public Livro {
```

O restante da declaração de classe é muito similar àquelas com as quais você tem trabalhado, com exceção da função construtora. Como pode ver, imediatamente após a declaração da construtora *FichaBiblioteca* está uma chamada da construtora *Livro*:

```
FichaBiblioteca(char *titulo, char *autor,
                char *editora) : Livro(titulo)
  {
    strcpy(FichaBiblioteca::autor, autor);
    strcpy(FichaBiblioteca::editora, editora);
  };
```

Quando o programa criar uma nova ocorrência da classe *FichaBiblioteca* (e, portanto, chama a construtora), a construtora *FichaBiblioteca* chamará primeiro a construtora para a classe-base (para *Livro*). Se você não especificar a construtora de classe-base após a construtora *FichaBiblioteca* como mostrado no fragmento de código a seguir, o compilador gerará um erro de sintaxe. Como você aprenderá em dicas posteriores, o compilador gerará o erro de sintaxe porque a construtora da classe *Livro* requer um único parâmetro. No caso de a construtora de classe *Livro* estar sem parâmetros, o compilador não irá requerer que você especifique a classe-base como mostrado no fragmento de código anterior.

1051 COMPREENDENDO AS CONSTRUTORAS BASE E DERIVADA

Ao derivar uma classe a partir de uma classe-base que tem uma função construtora, você precisa chamar a construtora de classe-base a partir de dentro da construtora da classe derivada. O programa a seguir, *basederi.cpp*, deriva uma classe chamada *Derivada* a partir da classe-base *Base*. Dentro da função construtora da classe *Derivada*, o código chama a construtora de classe *Base*, como a seguir:

```
#include <iostream.h>

class Base
 {
public:
   Base(void) { cout << "Construtora de classe-base\n"; };
 };

class Derivada:Base
 {
public:
   Derivada(void): Base()
     { cout << "Construtora de classe derivada\n"; };
 };

void main(void)
 {
   Derivada objeto;
 }
```

Cada vez que você criar uma ocorrência de uma classe derivada, o programa executará tanto a construtora para essa classe quanto a construtora para sua classe-base. Quando você compilar e executar o programa *basederi.cpp*, sua tela exibirá o seguinte:

```
Construtora de classe-base
Construtora de classe derivada
C:\>
```

Como vê, a construtora da classe-base é executada antes da construtora da classe derivada.

1052 PONDO OS MEMBROS PROTEGIDOS EM USO

Você aprendeu que C++ permite que seus programas declarem os membros de classe *protegidos*, que são totalmente acessíveis para as classes que seus programas derivam a partir de uma classe-base. O programa a seguir, *protege.cpp*, ilustra como usar os membros protegidos. No programa *protege.cpp*, a classe *Livro* define vários membros protegidos. Em seguida, o programa deriva a classe *FichaBiblioteca* a partir da classe *Livro*. Lembre-se, a classe derivada *FichaBiblioteca* pode acessar os membros protegidos dentro da classe *Livro*, o que permite que a chamada da função *exibe_custo* e a modificação do membro *custo* executem com sucesso. O código a seguir implementa o programa *protege.cpp*:

```
#include <iostream.h>
#include <string.h>

class Livro
 {
  public:
    Livro(char *titulo) { strcpy(Livro::titulo, titulo); };
    void exibe_titulo(void) { cout << titulo << endl; };
  protected:
    float custo;
    void exibe_custo(void) { cout << custo << endl; };
  private:
    char titulo[64];
```

```
};

class FichaBiblioteca : public Livro
 {
  public:
    FichaBiblioteca(char *titulo, char *autor, char *editora) : Livro(titulo)
      {
        strcpy(FichaBiblioteca::autor, autor);
        strcpy(FichaBiblioteca::editora, editora);
        custo = 49.95;
      };
    void exibe_biblio(void)
      {
        exibe_titulo();
        exibe_custo();
        cout << autor << ' ' << editora;
      };
  private:
    char autor[64];
    char editora[64];
 };

void main(void)
 {
   FichaBiblioteca ficha("Bíblia do Programador C/C++",
        "Jamsa e Klander", "Makron Books");
   ficha.exibe_biblio();
 }
```

Quando você compilar e executar o programa *protege.cpp*, sua tela exibirá a seguinte saída:

```
Bíblia do Programador C/C++
49.95
Jamsa e Klander    Makron Books
```

Como pode ver, os comandos de classe dentro da classe derivada *FichaBiblioteca* têm acesso completo aos membros protegidos dentro da classe-base *Livro*.

COMPREENDENDO QUANDO USAR OS MEMBROS PROTEGIDOS 1053

As classes derivadas podem acessar os membros de classe protegidos em sua classe-base. À medida que você cria suas classes, precisa decidir quais membros tornar públicos, privados ou protegidos. Como regra, você deve criar cada classe com a intenção de que uma classe derivada será usada mais tarde. Se você nunca usar a classe como uma classe-base, os membros protegidos serão essencialmente privados. Se você, mais tarde, decidir usar a classe como uma classe-base, predeterminando os membros protegidos poupará seu tempo de programação.

REVISANDO A HERANÇA DE CLASSE-BASE PÚBLICA OU PRIVADA 1054

Como você aprendeu, seus programas podem derivar uma segunda classe a partir de uma primeira classe. Até aqui, você derivou a segunda classe a partir da primeira classe como uma derivação pública, o que significa que os objetos da classe derivada podem acessar membros da classe-base. Para compreender isso melhor, considere o programa *baseder2.cpp*, como mostrado aqui:

```
#include <iostream.h>

class Base
```

```
 {
public:
   Base(void) { cout << "Construtora de classe-base\n"; };
   int dado;
 };

class Derivada:public Base
 {
public:
   Derivada(void): Base()   { cout << "Construtora de classe derivada \n"; };
 };

void main(void)
 {
   Derivada objeto;
   objeto.dado = 5;
   cout << objeto.dado << endl;
 }
```

Quando você compilar e executar o programa *baseder2.cpp*, o objeto que você criou a partir da classe *Derivada* poderá acessar diretamente o membro *dado* da classe *Base*. Como sabe, esse acesso direto viola as regras de encapsulamento e, portanto, deve ser evitado. Alternativamente, seu programa poderia herdar a classe *Base* como privada; que tem o efeito de tornar todos os membros da classe *Base* privados, como mostrado no programa *baseder3.cpp*:

```
#include <iostream.h>

class Base
 {
public:
   Base(void) { cout << "Construtora de classe-base\n"; };
   int dado;
 };

class Derivada: private Base
 {
public:
   Derivada(void): Base() { cout << "Construtora de classe derivada \n"; };
 };

void main(void)
 {
   Derivada objeto;
   objeto.dado = 5;
   cout << objeto.dado << endl;
 }
```

Quando o programa *baseder3.cpp* tenta acessar o membro *dado* da classe *Base*, o compilador falha e retorna um erro. No entanto, a classe *Derivada* ainda pode acessar a construtora da classe-base, de dentro do objeto. Adicionalmente, uma função de interface também permite que seu programa acesse o dado-membro. Na dica a seguir, você aprenderá sobre o terceiro método da herança da classe-base — herança de classe-base *protegida*.

1055 COMPREENDENDO A HERANÇA DE CLASSE-BASE PROTEGIDA

Ao criar classes dentro de seus programas, você poderá usar a palavra-chave *protected* para impedir que outras partes do programa acessem certos membros da classe, mas ainda tornar os membros da classe disponíveis para as classes herdadas. Você também pode herdar uma classe-base inteira como *protegida*. Quando você herda uma classe-base inteira como protegida, todos os membros públicos e protegidos da classe-base se tornam membros

protegidos da classe derivada. Para compreender melhor como as classes derivadas herdam os membros *públicos* e *protegidos*, considere o programa a seguir, *prot_bas.cpp*:

```
#include <iostream.h>

class base
 {
 protected:
   int i, j;
 public:
   void defij(int a, int b)
   {
     i = a;
     j = b;
   }
   void exibeij(void) {cout << i << " " << j << endl;}
 };

class derivada : protected base
 {
 private:
   int k;
 public:
   void setk(void)
   {
     defij(10,12);
     k = i * j;
   }
   void exibeall(void)
   {
     cout << k << " ";
     exibeij();
   }
 };

void main(void)
 {
   derivada objeto;
   // objeto.defij(2,3);  Este é um comando ilegal, pois
   //                     defij é um membro protegido de derivada.
   // objeto.exibeij();   Este também é um comando ilegal

   objeto.setk();
   objeto.exibeall();
 }
```

Como você pode ver, a classe *derivada* usa a palavra-chave *protected* para derivar a classe *base*. Fazer isso resulta em todos os membros da classe *base* serem *privados* dentro da classe *derivada*. Se o programa depois tentar acessar um membro herdado *privado* (tal como *defij* ou *exibeij*), o compilador retornará um erro porque esse membro está encapsulado. O código do programa observa que tanto as funções *objeto.defij(2,3)* e *objeto.exibeij()* são chamadas ilegais de função, porque os membros são protegidos dentro da classe derivada — o que significa que somente os objetos que podem acessar as funções que *derivada* herda de *base* são funções de interface ou classes que seu programa deriva a partir da classe *derivada*. Quando você compilar e executar o programa *prot_bas.cpp*, sua tela exibirá a seguinte saída:

```
120 10 12
C:\>
```

1056 COMPREENDENDO A HERANÇA MÚLTIPLA

Herança é a capacidade de uma classe herdar as características de outra classe. Herança múltipla é a capacidade de uma classe herdar as características de mais de uma classe-base. C++ suporta herança múltipla. Quando uma classe derivada herda características a partir de mais de uma classe-base, você simplesmente separa os nomes da classe-base usando vírgulas, como mostrado aqui:

```
class classe_derivada: public classe_base_1, public classe_base_2
 {
   public:
     // Membros públicos da classe derivada
   private:
     // Membros privados da classe derivada
 };
```

Da mesma forma, mais tarde, ao declarar a função construtora para a classe derivada, você precisará chamar a função construtora para cada classe-base. A Figura 1057 ilustra um uso simples de herança múltipla.

1057 UMA HERANÇA MÚLTIPLA SIMPLES

Você aprendeu que herança múltipla é a capacidade de uma classe derivada herdar as características de duas ou mais classes-base. Para muitos programadores, compreender como uma única classe pode herdar as características de duas outras classes pode ser confuso. O programa a seguir, *simpmult.cpp*, ilustra como a herança múltipla cria uma classe chamada *Livro*, que herda as classes-base *Pagina* e *Capa:*

```
#include <iostream.h>
#include <string.h>

class Capa
 {
public:
   Capa(char *titulo) { strcpy(Capa::titulo, titulo); };
 protected:
   char titulo[256];
 };

class Pagina
 {
 public:
   Pagina(int linhas = 55)  { Pagina::linhas = linhas; };
 protected:
   int linhas;
   char *texto;
 };

class Livro: public Capa, public Pagina {
 public:
   Livro(char *autor, char *titulo, float custo): Capa(titulo), Pagina(60)
   {
     strcpy(Livro::autor, autor);
     strcpy(Livro::titulo, titulo);
     Livro::custo = custo;
   };
   void exibe_Livro(void)
   {
      cout << titulo << endl;
      cout << autor << '\t' << custo;
   };
 private:
   char autor[256];
```

```
    float custo;
};

void main(void)
{
    Livro texto("Jamsa e Klander", "Bíblia do Programador C/C++", 49.95);
    texto.exibe_Livro();
}
```

O programa *simpmult.cpp* define duas classes, *Capa* e *Pagina*. Em seguida, o programa deriva uma terceira classe, *Livro*, a partir de ambas as classes originais. A construtora *Livro* passa parâmetros para as construtoras para as duas classes-base. Após o programa criar uma ocorrência do objeto *Livro*, ele gera saída com informações sobre o objeto. A função *exibe_Livro* exibe valores, um para cada uma das classes *Capa*, *Pagina* e *Livro*, como mostrado na seguinte saída:

```
Bíblia do Programador C/C++
Jamsa e Klander   49.95
C:\>
```

COMPREENDENDO A ORDEM DAS CONSTRUTORAS E AS CLASSES-BASE 1058

Herança múltipla é a capacidade de uma classe derivada herdar as características de mais de uma classe-base. Quando você usa a múltipla herança para derivar uma classe, a classe derivada precisa chamar as funções construtoras para cada classe-base. A ordem de chamada da construtora depende da ordem em que a classe derivada especificar as classes-base. Em outras palavras, se a classe derivada especificar suas classes-base como *Uma* e *Duas*, o programa chamará as construtoras na seguinte ordem: *Uma*, depois *Duas*, depois *derivada*, o que significa que seu código seria similar ao seguinte:

```
derivada(void) : Uma(), Duas(int i);
```

Para compreender melhor a ordem em que seus programas devem declarar os parâmetros da construtora, considere o programa a seguir, *multinv.cpp*, que ilustra a ordem da chamada da construtora quando você deriva uma classe a partir de três classes-base:

```
#include <iostream.h>

class Uma
{
 public:
   Uma(void)    { cout << "Construtora para Uma\n"; };
};

class Duas
{
 public:
   Duas(void)   { cout << "Construtora para Duas\n"; };
};

class Tres
{
 public:
   Tres(void)   { cout << "Construtora para Tres\n"; };
};

class Derivada: public Uma, public Tres, public Duas
{
 public:
   Derivada(void) : Uma(), Duas(), Tres()
   {
```

```
      cout << "Construtora Derivada chamada\n"; };
};

void main(void)
 {
   Derivada minha_classe;
 }
```

Quando você compilar e executar o programa *multinv.cpp*, sua tela exibirá a seguinte saída:

```
Construtora para Uma
Construtora para Tres
Construtora para Duas
Construtora Derivada chamada
C:\>
```

Como vê, o programa chama as funções construtoras na mesma ordem em que os nomes de classe-base aparecem no cabeçalho da classe:

```
class Derivada:  public Uma, public Tres, public Duas
```

1059 DECLARANDO UMA CLASSE-BASE COMO PRIVADA

Seus programas podem herdar uma única classe usando as palavras-chave *public, private* ou *protected.* Quando você deriva uma classe, pode preceder o nome da classe-base com *private* ou *public.* Quando seu programa usa a palavra-chave *public* para derivar uma classe-base, seu programa pode usar a classe derivada para acessar diretamente os membros públicos dentro da classe-base. No entanto, quando você usa a palavra-chave *private,* seu programa somente pode acessar os membros da classe-base por meio dos membros da classe derivada Finalmente, quando seus programas usam a palavra-chave *protected,* a classe que está derivando herda cada um dos membros públicos dentro da classe herdada como membros *protegidos,* o que permite que a classe derivada acesse os *membros da classe-base, permitindo também que outras classes derivem os membros protegidos. No entanto,* cada uma das dicas anteriores que ilustraram a herança múltipla usou a palavra-chave *public* diante dos nomes da classe-base, como mostrado aqui:

```
class Derivada: public Uma, public Tres, public Duas  {
```

Como veremos, seus programas poderão usar as palavras-chave *public, private* e *protected* quando seus programas efetuarem herança múltipla, exatamente como poderão fazer quando eles executarem a herança simples. O programa a seguir, *privmult.cpp*, deriva uma classe a partir de dois nomes de classe-base precedidos pela palavra-chave *private* e um nome de classe-base precedido pela palavra-chave *public*

```
#include <iostream.h>

class Uma
 {
 public:
  Uma(void)
  {
     cout << "Construtora para Uma\n";
     Uma = 1;
  };
  int Uma;
 };

class Duas
 {
 public:
  Duas(void)
  {
     cout << "Construtora para Duas\n";
     Duas = 2;
```

```
  };
  int Duas;
};

class Tres
{
public:
  Tres(void)
  {
    cout << "Construtora para Tres\n";
    Tres = 3;
  };
  int tres;
};

class Derivada: private Uma, private Tres, public Duas
{
public:
  Derivada(void) : Uma(), Duas(), Tres()
  {
    cout << "Construtora Derivada chamada\n"; };
  void exibe_valor(void)    { cout << Uma << Duas << Tres << endl; };
};

void main(void)
{
  Derivada minha classe;
  minha_classe.exibe_valor();
  cout << minha_classe.Duas;
}
```

Como a classe derivada declara a classe-base *Duas* como pública, o programa *privmult.cpp* pode acessar diretamente o membro *duas* sem ter de usar funções de interface. No entanto, o programa não pode acessar diretamente os valores *uma* ou *tres*, pois a classe derivada declara as classes-base *Uma* e *Tres* como privadas. Em vez disso, o programa precisa usar uma funçao-membro, tal como a função *exibe_valor*, para exibir os valores *uma* ou *tres*.

FUNÇÕES DESTRUTORAS E HERANÇA MÚLTIPLA 1060

Você aprendeu que, ao derivar uma classe a partir de uma classe-base, C++ chama a função construtora da classe-base antes de chamar a função construtora da classe derivada. No entanto, no caso das funções destrutoras, o oposto é verdadeiro: C++ chama a destrutora da classe derivada e depois chama as destrutoras para cada classe-base. O programa a seguir, *destrmult.cpp*, ilustra a seqüência de chamada das funções destrutoras de classe-base e derivada:

```
#include <iostream.h>

class Uma
{
 public:
   Uma(void)    { cout << "Construtora para Uma\n"; };
   ~Uma(void)   { cout << "Destrutora para Uma\n"; };
};

class Duas
{
 public:
   Duas(void)    { cout << "Construtora para Duas\n"; };
   ~Duas(void)   { cout << "Destrutora para Duas\n"; };
};
class Tres
{
 public:
```

```
    Tres(void)   { cout << "Construtora para Tres\n"; };
    ~Tres(void)  { cout << "Destrutora para Tres\n"; };
};

class Derivada: public Uma, public Duas, public Tres
{
public:
  Derivada(void) : Uma(), Duas(), Tres()
     { cout << "Construtora Derivada chamada\n"; };
  ~Derivada(void)
     { cout << "Destrutora Derivada chamada\n"; };
};

void main(void)
{
  Derivada minha_classe;
}
```

Quando você compilar e executar o programa *destmult.cpp*, sua tela exibirá a seguinte saída:

```
Construtora para Uma
Construtora para Duas
Construtora para Tres
Construtora Derivada chamada
Destrutora Derivada chamada
Destrutora para Tres
Destrutora para Duas
Destrutora para Uma
C:\>
```

Como vê, C++ chama as funções destrutoras na ordem oposta às funções construtoras.

1061 CONFLITOS DE NOMES ENTRE AS CLASSES BASE E DERIVADA

Quando você deriva uma nova classe usando uma ou mais classes-base, é possível que um nome de membro na classe derivada seja o mesmo que o nome de membro em uma ou mais das classes-base. Quando esses conflitos ocorrem, C++ usa o nome de membro da classe derivada. O programa a seguir, *conflito.cpp*, ilustra um conflito de nome de membro entre um nome de membro de classe-base e um nome de membro de classe derivada:

```
#include <iostream.h>

class Base
{
  public:
    void mostra(void)  { cout << "Esta é a classe-base" << endl; };
};

class Derivada: public Base
{
  public:
    void mostra(void)  { cout << "Esta é a classe Derivada" << endl; };
};

void main(void)
{
   Derivada minha_classe;
   minha_classe.mostra();
}
```

Quando você compilar e executar o programa *conflito.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a classe derivada
C:\>
```

Como a função-membro *mostra* é parte tanto da classe *Base* quanto da classe *Derivada*, existe um conflito potencial. No entanto, como a variável *minha_classe* é um objeto da classe *Derivada*, o compilador usará a versão *Derivada* da função *mostra*.

RESOLVENDO CONFLITOS DE NOME DE CLASSE E DE BASE 1062

A dica anterior mostrou que, quando um nome de membro de classe derivada está em conflito com um nome de membro de classe-base, C++ usa o nome de membro da classe derivada. No entanto, algumas vezes seu programa precisará acessar o membro da classe-base. Para fazer isso, seu programa pode usar o operador de resolução global (::). O programa a seguir, *resnome.cpp*, usa o operador de resolução global para acessar o membro *mostra* da classe-base:

```
#include <iostream.h>

class Base
 {
  public:
    void mostra(void)    { cout << "Esta é a classe-base" << endl; };
 };

class Derivada: public Base
 {
  public:
    void mostra(void) { cout << "Esta é a classe Derivada" << endl; };
 };

void main(void)
 {
   Derivada minha_classe;

   minha_classe.mostra();
   minha_classe.Base::mostra();
 }
```

Quando você compilar e executar o programa *resnome.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a classe derivada
Esta é a classe-base
C:\>
```

Como você viu na dica anterior, a referência simplesmente para a função *mostra* chamou a função da classe derivada. No entanto, como o segundo comando precede o nome da função com o operador de resolução global e o nome da classe *Base*, o segundo comando chama a função-membro *mostra* dentro da classe *Base*.

COMPREENDENDO QUANDO AS CLASSES HERDADAS EXECUTAM AS CONSTRUTORAS 1063

Você aprendeu que a função construtora da classe derivada sempre chamará a função construtora da classe a partir da qual ela derivar. No entanto, as classes derivadas também podem efetuar seu próprio processamento dentro da função construtora da classe derivada. Na verdade, quando seus programas derivam classes, cada ocorrência da classe chama a função construtora de toda classe na árvore hierárquica acima da classe derivada antes de executar a função construtora para a classe derivada. Por exemplo, suponha que sua classe *derivada5* seja derivada de quatro outras classes derivadas e uma classe *base*. Se cada construtora de classe exibir uma mensagem, quando você criar qualquer ocorrência da classe *derivada5*, suas construtoras exibirão a seguinte mensagem:

```
Construindo a classe de base.
Construindo a classe derivada1.
Construindo a classe derivada2.
Construindo a classe derivada3.
Construindo a classe derivada4.
Construindo a classe derivada5.
```

Como você aprendeu, o programa executa as funções destrutoras na seqüência inversa às funções construtoras, de modo que elas exibirão uma série de mensagens similar à seguinte:

```
Destruindo a classe derivada5.
Destruindo a classe derivada4.
Destruindo a classe derivada3.
Destruindo a classe derivada2.
Destruindo a classe derivada1.
Destruindo a classe de base.
```

1064 EXEMPLO DE UMA CONSTRUTORA DE CLASSE HERDADA

Os programas C++ chamam a construtora e destrutora da classe para cada classe acima de uma classe derivada na árvore hierárquica. Para compreender melhor como C++ chama as funções construtoras e destrutoras para as classes derivadas, considere o programa *cons_des.cpp*. O programa *cons_des.cpp* constrói e destrói um único objeto do tipo *derivada2* e chama as construtoras e destrutoras a partir das classes acima de *derivada2* na árvore, como mostrado aqui:

```
#include <iostream.h>

class base
 {
  public:
    base(void)   { cout << "Construindo base.\n"; }
    ~base(void)  { cout << "Destruindo base.\n";  }
 };

class derivada1 : public base
 {
  public:
    derivada1(void)  { cout << "Construindo derivada1.\n"; }
    ~derivada1(void) {cout << "Destruindo derivada1.\n";}
 };

class derivada2 : public derivada1
 {
  public:
    derivada2(void)  { cout << "Construindo derivada2.\n"; }
    ~derivada2(void) {cout << "Destruindo derivada2.\n";}
 };

void main(void)
 {
   derivada2 objeto;
 }
```

Quando você compilar e executar o programa *cons_des.cpp*, sua tela exibirá a seguinte saída:

```
Construindo base.
Construindo derivada1.
Construindo derivada2.
Destruindo derivada2.
Destruindo derivada1.
Destruindo base.
C:\>.
```

COMO PASSAR PARÂMETROS PARA AS CONSTRUTORAS DE CLASSE-BASE 1065

Como sabe, toda vez que você criar uma ocorrência de uma classe derivada, seu programa chamará a construtora para cada classe a partir da qual a classe derivada deriva, além da construtora de classe derivada. Em muitos casos, as classes-base a partir das quais a classe deriva incluirão funções construtoras que esperam parâmetros. Se sua classe derivada não passar parâmetros para as funções construtoras das classes mais elevadas na árvore hierárquica (isto é, a classe-base a partir da qual a classe deriva), um erro ocorrerá e o programa não será compilado. Portanto, C++ permite e espera que você passe parâmetros para as funções construtoras acima de um objeto na árvore. Para suportar as declarações para a classe-base ou classes acima da classe derivada na árvore, você pode usar a forma geral expandida da declaração de construtora de classe derivada, como mostrado aqui:

```
derivada construtora(lista-arg): base1(lista-argumento);
                                 base2(lista-argumento);
                                 base3(lista-argumento);
                                   .
                                   .
                                   .
                                 baseN(lista-argumento);
 {
   // Corpo da construtora derivada
 }
```

Na declaração precedente, *base1* até *baseN* representa as classes acima da classe derivada dentro da árvore hierárquica. Para compreender melhor como você chamará as construtoras de classe-base para uma classe derivada, considere o programa a seguir, *cls_parm.cpp*:

```
#include <iostream.h>

class base
 {
  protected:
    int i;
  public:
    base(int x)
    {
      i=x;
      cout << "Construindo base.\n";
    }
    ~base(void)  {cout << "Destruindo base.\n";}
};

class derivada : public base
 {
   int j;
  public:
    // derivada usa x; base usa y.
    derivada(int x, int y): base(y)
      {
        j = x;
        cout << "Construindo derivada.\n";
      }
    ~derivada(void) {cout << "Destruindo derivada.\n";}
    void show(void) {cout << i << ", " << j << endl;}
};

void main(void)
 {
   derivada object(3,4);
```

```
    object.show();                  // Exibe 4, 3
}
```

O programa *cls_parm.cpp* deriva a classe *derivada* a partir da classe *base*. No início da execução, o programa cria uma ocorrência da classe *derivada* e passa o valor 3 e 4 para a construtora. A construtora *derivada* passa o segundo valor (neste caso, 4) para a construtora *base*. Após a construtora *base* completar seu processamento, a construtora *derivada* usa o primeiro valor (neste caso, 3) como parâmetro. Quando você compilar e executar o programa *cls_parm.cpp*, sua tela exibirá a seguinte saída:

```
Construindo base.
Construindo derivada.
4, 3
Destruindo derivada.
Destruindo base.
C:\>
```

1066 COMPREENDENDO AS DECLARAÇÕES DE ACESSO COM AS CLASSES DERIVADAS

Na Dica 1054 você aprendeu que quando uma classe derivada herda uma classe-base como privada, todos os membros públicos e protegidos dessa classe se tornam membros privados da classe derivada — o que significa que os membros são encapsulados dentro da classe, e seus programas somente podem acessar os membros por meio das funções de interface da classe derivada. No entanto, em certas circunstâncias, você poderá querer restaurar especificações de acesso originais de um ou mais membros herdados. Por exemplo, você poderia querer conceder a certos membros *públicos* da classe-base status *public* na classe derivada, mesmo que a classe derivada tenha herdado a classe-base como *private*. Para fazer isso, você precisa usar uma declaração de acesso dentro da classe derivada. As declarações de acesso dentro de seus programas terão a seguinte forma:

```
classe-base::membro;
```

Para compreender melhor como funciona uma declaração de acesso, considere o seguinte fragmento de código.

```
class base
 {
   public:
      int j;
      int k;
 };
class derivada : private base
 {
   public:
      base::j;
      // Mais declarações aqui
 };
```

No exemplo anterior, a variável *j* normalmente seria privada dentro da classe *derivada* (pois a classe *derivada* usou a palavra-chave *private* para herdar a classe-*base*). No entanto, o comando *base::j*, que redeclara *j* como *public* dentro da classe *derivada*, retorna *j* ao status público, sem causar impacto ao restante da classe. Por outro lado, a variável *k* permanecerá privada dentro da classe *derivada*.

1067 USANDO DECLARAÇÕES DE ACESSO COM AS CLASSES DERIVADAS

Na dica anterior você aprendeu sobre as declarações de acesso dentro de classes derivadas. No pequeno fragmento de código dentro dessa dica, você aprendeu os fundamentos sobre o uso das declarações de acesso. O programa a seguir, *ac_decl.cpp*, usa várias declarações adicionais para levar o conceito de declarações de acesso mais adiante. No programa *ac_decl.cpp*, o código converte três funções *privadas* dentro da classe *derivada* de volta ao acesso *público*. No entanto, como mostra a última declaração (dentro de um comentário), o programa não pode

declarar o membro *i* de volta como *público*, porque ele é *privado* dentro da classe-*base*. Tornar *i público* dentro da classe *derivada* violaria as regras de encapsulamento, e, portanto, o compilador retornaria um erro na atribuição. O programa *ac_decl.cpp* fornece uma boa idéia sobre como as declarações de acesso funcionam:

```
#include <iostream.h>

class base
 {
   int i;          // privado na classe base
  public:
    int j, k;
    void def_i(int x) {i = x;}
    int obtem_i(void) {return i;}
 };

class derivada : private base
 {
  public:
                    // Os próximos comandos anulam a herança privada.
    base::j;        // torna j público novamente
    base::def_i;    // torna def_i() público
    base::obtem_i;  // torna obtem_i() público
    // base::i; é um comando ilegal, você não pode remover o acesso.
    int a;
 };

void main(void)
 {
   derivada objeto;

   //objeto.i = 10; Comando ilegal, i é privado para base.
   objeto.j = 20;   // legal, pois j é público

   //objeto.k = 30; Ilegal porque k é privado para derivada
   objeto.a = 40;
   objeto.def_i(10);
   cout << objeto.obtem_i() << ", " << objeto.j << ", " << objeto.a;
 }
```

Como você pode ver, o programa *ac_decl.cpp* herda a classe *derivada* usando a palavra-chave *private* e, depois, usa muito a classe *derivada* pública novamente. Como regra geral, ao usar declarações de acesso dentro de suas classes derivadas, você deverá manter o número de declarações de acesso a um mínimo, ou reconsiderar como escreveu seu código. Manter as declarações de acesso a um mínimo evita confusão (tanto sua quanto de outros programadores que forem ler o seu código) e torna seus programas mais consistentes com as regras de encapsulamento.

EVITANDO AMBIGÜIDADE COM AS CLASSES-BASE VIRTUAIS 1068

Quando seu programa deriva uma classe que herda múltiplas classes derivadas anteriormente a partir de uma única classe-base, é possível que a classe possa incluir membros que, embora únicos para cada uma das classes-pai, compartilhem o mesmo nome dentro da classe derivada. Quando essa situação ocorrer, a ambigüidade do membro de classe fará o compilador falhar. Por exemplo, se você escrever um programa que usa a classe *base* e derivar dela duas classes, *derivada1* e *derivada2*, seu programa não terá ambigüidade. No entanto, se seu programa mais tarde derivar a classe *derivada3* a partir, tanto da *derivada1* quanto da *derivada2*, cada objeto da classe *derivada3* na verdade conterá dois objetos da classe *base*, como mostrado na Figura 1068.

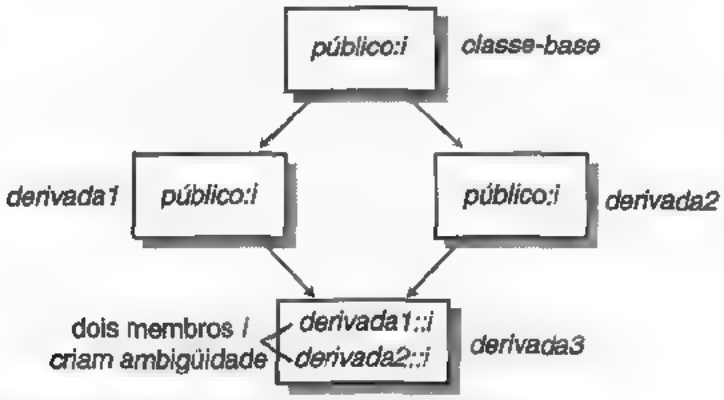

Figura 1068 Derivar a partir de múltiplas classes pode criar ambigüidade.

Não somente é desnecessário para cada objeto da classe *derivada3* conter dois objetos da classe *base*, como tal ambigüidade provavelmente confundirá você e o compilador. Por exemplo, o programa a seguir, *ambig_cl.cpp*, cria ambigüidade na classe *derivada*. Por causa da ambigüidade, o compilador não completará sua execução, e, em vez disso, retornará um erro, o que forçará o compilador a parar com um erro·

```
// Este programa contém erros e não passa na compilação.
#include <iostream.h>

class base
 {
  public:
    int i;
 };

class derivada1 : public base
 {
  public:
    int j;
 };

class derivada2 : public base
 {
  public:
    int k;
 };

class derivada3 : public derivada1, public derivada2
 {
  public:
    int soma;
 };

void main(void)
 {
   derivada3 objeto;

   objeto.i = 10; // Isso faz o compilador parar por não
                  // saber qual i você está referenciando
   objeto.j = 20;
   objeto.k = 30;
   objeto.soma = objeto.i + objeto.j + objeto.k;
   cout << objeto.i << " ";
   cout << objeto.j << " " << objeto.k << " ";
   cout << objeto.soma << endl;
 }
```

Como a classe *Base* inclui o membro público *i*, e tanto as classes *derivada1* quanto *derivada2* derivam a partir da classe *Base*, a classe *derivada3* na verdade inclui duas ocorrências do membro público *i*. Como o compilador não pode resolver a ambigüidade do membro *i*, o programa *ambig_cl.cpp* retorna três erros de compilador — todos provenientes do uso ambíguo do membro *i* dentro da variável-objeto *derivada3*. Como você aprenderá na dica a seguir, seus programas podem usar classes-base virtuais para evitar os erros de ambiguidade dentro das classes derivadas.

COMPREENDENDO AS CLASSES-BASE VIRTUAIS 1069

Vimos na dica anterior que, quando seus programas derivam dois ou mais objetos a partir de uma classe-base comum, eles podem encontrar erros de ambiguidade. Para evitar a ambiguidade, você precisará evitar ter múltiplas cópias da classe-base dentro de um dado objeto. Para evitar que seus programas herdem múltiplas cópias de uma dada classe-base, você pode declarar essa classe-base usando a palavra-chave *virtual*, como mostrado no programa a seguir, *clasvirt.cpp:*

```
#include <iostream.h>

class base
 {
  public:
    int i;
 };

class derivada1 : virtual public base
 {
  public:
    int j;
 };

class derivada2 : virtual public base
 {
  public:
    int k;
 };

class derivada3 : public derivada1, public derivada2
 {
  public:
    int soma;
 };

void main(void)
 {
   derivada3 objeto;

   objeto.i = 10;              // agora i não é ambígua
   objeto.j = 20;
   objeto.k = 30;
   objeto.soma = objeto.i + objeto.j + objeto.k;
   cout << objeto.i << " ";
   cout << objeto.j << " " << objeto.k << " ";
   cout << objeto.soma << endl;
 }
```

No caso do programa *clasvirt.cpp*, o fato de cada uma das duas classes derivadas intermediárias herdar a classe-base virtualmente, permitirá que a classe derivada final (*derivada3*) herde ambas as classes sem preocupação com a ambiguidade. Como você viu na dica anterior, sem a palavra-chave *virtual*, a classe *derivada3* efetivamente tem dois membros *i*. Como regra, se seus programas derivam classes a partir de múltiplas classes compartilhando uma única classe-mãe, seus programas devem usar a palavra-chave *virtual* na definição de classe para evitar ambiguidade.

1070 AMIGAS MÚTUAS

Você aprendeu que C++ lhe permite especificar outras classes como amigas de uma classe, o que permite que as funções na classe *amiga* acessem os métodos privados de uma classe. À medida que você examinar programas C++, poderá encontrar casos em que duas classes são *amigas mútuas*. Em outras palavras, as funções em uma classe podem acessar os dados privados de outra classe, e as funções na outra classe podem acessar os dados privados dessa classe. O programa a seguir, *mutuas.cpp*, ilustra duas classes que são amigas mútuas:

```
#include <iostream.h>
#include <string.h>

class Juca
 {
  public:
    Juca(char *msg) { strcpy(mensa, msg); };
    void exibe_mensa(void)    { cout << mensa << endl; };
    friend class Zeca;
    void exibe_zeca(class Zeca zeca);
  private:
    char mensa[256];
 };

class Zeca
 {
  public:
    Zeca(char *msg) { strcpy(mensa, msg); };
    void exibe_mensa(void)    { cout << mensa << endl; };
    friend class Juca;
    void exibe_juca(class Juca juca);
  private:
    char mensa[256];
 };

void Juca::exibe_zeca(class Zeca zeca) {  cout << zeca.mensa << endl;  };

void Zeca::exibe_juca(class Juca juca) {  cout << juca.mensa << endl;  };

void main(void)
 {
   class Zeca zeca("Oba, oba, oba...");
   class Juca juca("Epa, epa, epa...");

   zeca.exibe_mensa();
   zeca.exibe_juca(juca);
   juca.exibe_mensa();
   juca.exibe_zeca(zeca);
 }
```

Como o programa *mutuas.cpp* usa o qualificador *friend* ao declarar ambas as classes, as funções amigas mútuas *Juca* e *Zeca* podem acessar os dados privados uma da outra. Quando você compilar e executar o programa *mutuas.cpp*, sua tela exibirá a seguinte saída:

```
Oba, oba, oba...
Epa, epa, epa...
Epa, epa, epa...
Oba, oba, oba...
C:\>
```

COMO UMA CLASSE DERIVADA PODE TORNAR-SE UMA CLASSE-BASE 1071

C++ lhe permite criar uma hierarquia de herança, que permite a uma classe herdar as características de uma classe-base que pode ela mesma ter herdado características de uma terceira classe-base. O programa a seguir, *tresniv.cpp*, deriva três níveis de classes. Como você pode ver, cada classe sucessiva herda as características de cada classe antes dela:

```
#include <iostream.h>

class Base
 {
 public:
   void exibe_base(void)  { cout << "Mensagem da classe-base\n"; };
 };

class Nivel1 : public Base
 {
  public:
    void exibe_nivel1(void)
    {
       exibe_base();
       cout << "Mensagem do nível 1\n";
    };
 };

class Nivel2 : public Nivel1
 {
  public:
    void exibe_nivel2(void)
     {
        exibe_nivel1();
        cout << "Mensagem do nível 2\n";
     };
 };

class Nivel3 : public Nivel2
 {
  public:
    void exibe_nivel3(void)
     {
        exibe_nivel2();
        cout << "Mensagem do nível 3\n";
     };
 };

void main(void)
 {
   Nivel3 meus_dados;

   meus_dados.exibe_nivel3();
 }
```

Como você pode ver a partir do código para o programa *tresniv.cpp*, a classe *Nivel3* deriva a partir de três classes anteriores. Para compreender como *Nivel3* deriva a partir de todas as três classes anteriormente definidas, considere cada derivação em ordem. A classe *Nivel3* deriva diretamente a partir da classe *Nivel2*. Por sua vez, a classe *Nivel2* deriva diretamente da classe *Nivel1*. Finalmente, a classe *Nivel1* deriva a partir da classe *Base*. Quando você compilar e executar o programa *tresniv.cpp*, sua tela exibirá a seguinte saída:

```
Mensagem da classe-base
Mensagem no nível 1
Mensagem no nível 2
Mensagem no nível 3
C:\>
```

1072 USANDO MEMBROS PROTEGIDOS NAS CLASSES DERIVADAS

Cada vez que você cria uma nova classe, deve assumir que a classe poderá eventualmente se tornar a classe-base para outras derivações de classe. Portanto, você deve utilizar os membros protegidos para limitar o acesso do programa aos membros da classe, tanto para ocorrências da classe atual quanto ocorrências de quaisquer classes futuras derivadas. Como você aprendeu, os membros protegidos permitem que a classe derivada acesse os membros de uma classe-base como se esses membros fossem *publicos*, mas não permitam derivações além do primeiro acesso a membros da classe-base como membros *públicos*. O programa a seguir, *protderi.cpp*, ilustra como usar dados protegidos em uma classe derivada, o que, por sua vez, se tornou uma classe-base:

```
#include <iostream.h>
#include <string.h>

class Base
 {
  public:
    Base(char *str)    { strcpy(mensa, str); };
    void exibe_base(void)    { cout << mensa << endl; };
  protected:
    char mensa[256];
 };

class Nivel1 : public Base
 {
  public:
    Nivel1(char *str, char *base) : Base(base)  { strcpy(mensa, str);};
    void exibe nivel1(void)  { cout << mensa << endl; } ;
  protected:
    char mensa[256];
 };

class Menor : public Nivel1
 {
  public:
    Menor(char *str, char *nivel1, char *base) : Nivel1(nivel1, base)
      { strcpy(mensa, str); };
    void exibe_menor(void)
     {
       exibe_base();
       exibe nivel1();
       cout << mensa << endl;
     };
  protected:
    char mensa[256];
 };

void main(void)
 {
   Menor piso("Mensagem menor", "Mensagem do Nível1", "Mensagem da Base");

   piso.exibe_menor();
 }
```

Como você pode ver no código, cada classe define o membro *mensa* como protegido — o que permite que as classes derivadas derivem esse membro, mas evita que o código de fora modifique o membro *mensa* diretamen-

te. Como cada classe derivada define o membro *mensa* como *protegido*, as classes derivadas podem acessar diretamente os dados se quiserem. No entanto, no programa *protderi.cpp*, as classes derivadas ainda usam as funções de interface. Como regra, seus programas devem impor os princípios de encapsulamento, mesmo ao trabalhar com membros protegidos a partir de uma classe derivada. Quando você compilar e executar o programa *protderi.cpp*, sua tela exibirá a seguinte saída:

```
Mensagem da classe-base
Mensagem do nível 1
Mensagem menor
C:\>
```

DEFININDO DADOS DE CLASSE ESTÁTICA 1073

Você aprendeu que, ao declarar membros de classe, C++ lhe permite preceder uma definição com o qualificador *static*. Por exemplo, a seguinte definição de classe usa *static* e dados-membro não-estáticos.

```
class AlgumaClasse
 {
  public:
    static int conta;
    AlgumaClasse(int valor)
    {
      conta++;
      meus_dados = valor;
    };
    ~AlgumaClasse(void)  { conta --;  };
    int meus_dados;
 };
```

Normalmente, cada ocorrência de objeto recebe seus próprios dados-membro. No entanto, se você preceder uma definição de membro com a palavra-chave *static*, todas as ocorrências de objeto compartilharão o membro. Se uma ocorrência mudar os dados, todas as ocorrências imediatamente reconhecerão o novo valor do membro modificado. A definição de um membro estático não aloca memória para o membro. Em vez disso, você precisa declarar a variável *estática* fora da classe, como mostrado aqui:

```
int AlgumaClasse::conta;
```

O programa a seguir, *compart.cpp*, usa a palavra-chave *static* para compartilhar a variável-membro *conta* (que controla o número de ocorrências de objetos):

```
#include <iostream.h>

class AlgumaClasse
 {
  public:
    static int conta;
    AlgumaClasse(int valor)
    {
      conta++;
      meus_dados = valor;
    };
    ~AlgumaClasse(void) {   conta--;   };
    int meus_dados;
 };

int AlgumaClasse::conta;

void main(void)
 {
   AlgumaClasse Uma(1);
   cout << "Uma: " << Uma.meus_dados << ' ' << Uma.conta  << endl;
```

```
    // Declara outra ocorrência
    AlgumaClasse Duas(2);
    cout << "Duas: " << Duas.meus_dados << ' ' << Duas.conta << endl ;

    // Declara outra ocorrência
    AlgumaClasse Tres(3);
    cout << "Tres: " << Tres.meus_dados << ' ' << Tres.conta << endl ;
  }
```

Toda vez que o programa cria uma nova ocorrência de objeto, a construtora da classe incrementa a variável *estática conta*. Após três ocorrências, *conta* contém o valor 3. Como você pode ver, todas as três ocorrências compartilham o membro *estático*. Quando você compilar e executar o programa *compart.cpp*, sua tela exibirá a seguinte saída:

```
Uma: 1  1
Duas: 2  2
Tres: 3  3
C:\>
```

1074 INICIALIZANDO UM DADO-MEMBRO ESTÁTICO

Você aprendeu na dica anterior que C++ lhe permite declarar dados-membro *estáticos* que são acessíveis a todas as ocorrências da classe. Quando você usa dados-membro *estáticos*, precisa determinar o melhor modo de inicializar os membros. Um modo é permitir que a primeira ocorrência passe o valor desejado para a função construtora. Para fazer isso, você sobrecarrega a função construtora para que suporte um ou dois parâmetros. Se o inicializador do objeto passar dois parâmetros, a função construtora atribuirá o segundo parâmetro à variável *estática*. O programa a seguir, *esta_ini.cpp*, sobrecarrega a função construtora desse modo para inicializar o membro *estático* com 999:

```
#include <iostream.h>

class AlgumaClasse
 {
  public:
    static int conta;
    AlgumaClasse(int valor)
    {
      conta++;
      meus_dados = valor;
    };
    AlgumaClasse(int valor, int valor_estatico)
     {
       conta = valor_estatico;
       meus_dados = valor;
     };
    ~AlgumaClasse(void) { conta--; };
    int meus_dados;
 };

int AlgumaClasse::conta;

void main(void)
 {
   AlgumaClasse Uma(1, 999);
   cout << "Uma: " << Uma.meus_dados << ' ' << Uma.conta  << endl;

   // Declara outra ocorrência
   AlgumaClasse Duas(2);
   cout << "Duas: "   << Duas.meus_dados << ' ' << Duas.conta << endl ;

   // Declara outra ocorrência
```

```
    AlgumaClasse Tres(3);
    cout << "Tres: " << Tres.meus_dados << ' ' << Tres.conta << endl ;
  }
```

Como você aprenderá na dica a seguir, seus programas também podem acessar diretamente um membro *estático público* para atribuir ou referenciar o valor do membro. Quando você compilar e executar o programa *esta_ini.cpp*, sua tela exibirá a seguinte saída:

```
Uma: 1 999
Duas: 2 1000
Tres: 3 1001
C:\>
```

ACESSO DIRETO DE UM DADO-MEMBRO ESTÁTICO 1075

Na dica anterior você sobrecarregou uma função construtora para ajudar seus programas a inicializar um dados-membro *estático*. Quando um dado-membro *estático* é público, seus programas podem acessar diretamente o valor do membro. Portanto, o programa *esta_ini.cpp* poderia ter inicializado o membro usando duas técnicas diferentes. Primeiro, o programa poderia ter atribuído o valor quando declarou o membro fora da classe, como a seguir:

```
int AlgumaClasse::conta = 999;
```

Segundo, dentro de *main*, o programa poderia ter acessado o membro estático diretamente, como mostrado aqui:

```
void main(void)
  {
    AlgumaClasse::conta = 999;

    AlgumaClasse Uma(1);

    // Outros comandos
  }
```

Quando você declarar um membro *estático público*, o programa poderá acessar diretamente o valor do membro *mesmo se* nenhuma ocorrência da classe existir. Para proteger melhor os membros *estáticos*, use membros de dados *estáticos privados*, como discutido na Dica 1076, a seguir.

COMPREENDENDO OS DADOS-MEMBRO PRIVADOS ESTÁTICOS 1076

Como você aprendeu, C++ lhe permite declarar membros de classe *estáticos* que todas as ocorrências de uma classe podem acessar. Se o membro também for *público*, o programa poderá acessar o membro, ignorando as ocorrências de classes. Para compreender melhor o membro *estático*, você poderá declará-lo como *privado*. Quando o membro *estático* é *privado*, somente funções-membro da classe podem acessá-lo. O programa a seguir, *privest.cpp*, ilustra como usar um membro *privado estático*:

```
#include <iostream.h>

class AlgumaClasse  {
  public:
    AlgumaClasse(int valor)
              {
                conta++;
                meus_dados = valor;
              };
    AlgumaClasse(int valor, int valor_estatico)
    {
       conta = valor_estatico;
```

```
        meus_dados = valor;
    };
    ~AlgumaClasse(void) { conta--; };
    void exibe_valores(void) { cout << meus_dados << ' '
                               << conta << endl; };
  private:
    static int conta;
    int meus_dados;
};

int AlgumaClasse::conta;

void main(void)
{
  AlgumaClasse Uma(1, 999);
  Uma.exibe_valores();

  // Declara outra ocorrência
  AlgumaClasse Duas(2, 1000);
  Duas.exibe_valores();

  // Declara outra ocorrência
  AlgumaClasse Tres(3);
  Tres.exibe_valores();
}
```

Quando você declara um membro *privado estático*, como mostrado no programa *privest.cpp*, seu programa pode usar uma construtora para inicializar os membros, ou pode atribuir um valor na inicialização que aparece fora da definição da classe.

1077 COMPREENDENDO AS FUNÇÕES-MEMBRO ESTÁTICAS

C++ lhe permite usar dados-membro de classe *estática* cujos valores são compartilhados por uma a uma das ocorrências. Além de suportar dados-membro *estáticos*, C++ também suporta funções-membro *estáticas*. No entanto, os programadores C++ normalmente não usam funções-membro *estáticas*. Em geral, o único uso para as funções-membro *estáticas* é manipular dados-membro *estáticos*. Ao contrário de outras funções-membro, que podem usar o ponteiro *this* para acessar dados da ocorrência, as funções-membro *estáticas* não podem acessar o ponteiro *this* ou dados da ocorrência. Portanto, a única vez que você usará uma função-membro *estática* será quando tiver uma função que não manipula dados da ocorrência. O programa a seguir, *estatica.cpp*, ilustra como usar uma função-membro *estática*:

```
#include <iostream.h>

class AlgumaClasse
{
  public:
    AlgumaClasse(int valor)  { algum_valor = valor; };
    void exibe_dados(void)  { cout << dado << ' ' << algum_valor << endl; };
    static void def_dados(int valor) { dado = valor; };
  private:
    static int dado;
    int algum_valor;
};

int AlgumaClasse::dado;

void main(void)
{
  AlgumaClasse minha_classe(1001);
  minha_classe.def_dados(5005);
  minha_classe.exibe_dados();
}
```

No programa *estatica.cpp*, tanto o membro *dado* quanto o membro *def_dado* são membros estáticos. Embora o programa *estatica.cpp* crie somente uma única ocorrência do objeto *AlgumaClasse*, ele poderia facilmente criar mais — e todas compartilhariam o mesmo valor estático, 5005.

ACESSO DIRETO DE UMA FUNÇÃO ESTÁTICA PÚBLICA 1078

Como você aprendeu na dica anterior, C++ lhe permite definir funções *estáticas públicas* dentro de uma classe. Quando você declara essas funções como *públicas*, elas são totalmente acessíveis em todo o programa, mesmo se o programa ainda não tiver criado uma ocorrência da classe. Para acessar uma função-membro *estática pública*, seu programa usará o operador de resolução global (::), como mostrado aqui:

```
nome_classe::nome_membro(parâmetros);
```

O programa a seguir, *globesta.cpp*, ilustra o acesso direto de uma função-membro *estática pública*. Observe que o programa usa a função *mensa*, embora não existam ocorrências de um objeto *AlgumaClasse*:

```
#include <iostream.h>

class AlgumaClasse
 {
  public:
    static void mensa(void) {cout << "Oi, pessoal!\n"; );
 };

void main(void)
 {
   AlgumaClasse::mensa();
 }
```

USANDO TIPOS EXPANDIDOS COMO MEMBROS DA CLASSE 1079

Por simplicidade, a maioria dos exemplos apresentados nesta seção usou membros de classe que eram valores do tipo *int*, *float* e *char*. No entanto, à medida que suas definições de classe se tornarem mais complexas, seus membros de classe poderão ser ponteiros, referências, tipos enumerados e até classes embutidas. O programa a seguir, *mbr_belo.cpp*, ilustra como usar membros de classe mais complexos:

```
#include <iostream.h>

enum Dias { Segunda, Terca, Quarta, Quinta, Sexta };

class ClasseBela
 {
  public:
    int *numero_sorte;
    enum Dias dia_sorte;
 };

void main(void)
 {
   ClasseBela waal;
   int sorte = 1500;

   waal.dia_sorte = Segunda;
   waal.numero_sorte = &sorte;
   cout << "Meu número de sorte é " << *(waal.numero_sorte)  << endl;
   switch (waal.dia_sorte)
   {
     case Segunda: cout <<"Meu dia de sorte é Segunda\n";
                   break;
     default: cout << "Meu dia de sorte não é qualquer dia, somente
```

```
        Segunda\n";
    };
}
```

A classe *ClasseBela* define um único membro público do tipo ponteiro *int*, e um único membro do tipo *Dias* enumerado definido imediatamente antes da definição de *ClasseBela* dentro do arquivo do programa. Quando o programa é executado, ele declara uma ocorrência da *ClasseBela* chamada *waal* e uma variável *int* chamada *sorte*. O programa atribui ao membro *dia_sorte* o valor enumerado *Segunda*, e atribui ao membro *numero_sorte* uma referência à variável *sorte*. Em seguida, o programa gera saída com base nos valores que atribui aos membros *ClasseBela*. Quando você compilar e executar o programa *mbr_belo.cpp*, sua tela exibirá a seguinte saída:

```
Meu número de sorte é 1500
Meu dia de sorte é Segunda
C:\>
```

1080 EMBUTINDO UMA CLASSE DENTRO DE OUTRA

Na dica anterior vimos que C++ permite que os membros de suas classes sejam de qualquer tipo, incluindo outras classes. O programa a seguir, *embutida.cpp*, ilustra como usar uma classe embutida dentro de outra:

```
#include <iostream.h>

class Externa
 {
  public:
   Externa(void)
     { cout << "Gerada a ocorrência de uma externa\n";
       dados_externos = 2002;
     };
   class Interna
   {
    public:
      Interna(void)
       { :
          cout << "Gerada a ocorrência de uma interna\n";
          dados_internos = 1001;
       };
      void exibe_dados(void)  { cout << "Interna: " << dados_internos
      << endl; };
    private:
      int dados_internos;
   } coisas_dentro;
   void exibe_todos_dados(void)
   {
      coisas_dentro.exibe_dados();
      cout << "Externa: " << dados_externos << endl;
   };
 private:
   int dados_externos;
 };

void main(void)
 {
   Externa meus_dados;
   meus_dados.exibe_todos_dados();
 }
```

Quando você compilar e executar o programa *embutida.cpp*, sua tela exibirá o seguinte:

```
Gerada a ocorrência de uma interna
Gerada a ocorrência de uma externa
Interna: 1001
```

```
Externa: 2002
C:\>
```

Como regra, as classes embutidas podem se tornar muito difíceis de compreender e podem reduzir a reutilização do programa. Uma solução melhor seria implementar duas classes distintas, usando a classe *Interna* como uma classe-base a partir da qual você possa derivar a classe *Externa*.

COMPREENDENDO SUBCLASSES E SUPERCLASSES 1081

Lendo outros artigos e livros sobre C++ ou Java, outra linguagem de programação orientada a objetos, você poderá encontrar os termos *subclasse* e *superclasse*. Os termos relacionam-se à herança de classe. Em geral, os termos *subclasse* e *classe-base* são intercambiáveis. Da mesma forma, os termos *superclasse* e *classe derivada* também são intercambiáveis. Para suas discussões em C++, adote os termos *classe-base* e *classe derivada*.

COMANDOS EM LINGUAGEM ASSEMBLY IN-LINE EM UMA FUNÇÃO MÉTODO 1082

Como você aprendeu, muitos compiladores C e C++ permitem que seus programas coloquem comandos em linguagem assembly in-line dentro do código do seu programa. Como você aprenderá nesta dica, seus programas também podem colocar comandos em linguagem assembly dentro dos métodos da classe. O programa a seguir, *clasbipa.cpp*, cria dois membros, *bipa* e *bipabipa*, que usam comandos da linguagem assembly in-line para soar o alto-falante interno do computador:

```
#include <iostream.h>

class Alarmes
 {
 public:
   void bipa(void);
   void bipabipa(void);
 };

void Alarmes::bipa(void)
 {
  asm
  {
    mov ah,2;
    mov dl,7;
    int 0x21;
  }
 }

void Alarmes::bipabipa(void)
 {
  asm
  {
     mov ah,2;
     mov dl,7;
     int 0x21;
     mov ah,2;
     mov dl,7;
     int 0x21;
  }
 }

void main(void)
 {
   Alarmes ruido;
```

```
    ruido.bipa();
    ruido.bipabipa();
}
```

Quando você usar linguagem assembly in-line, a maioria dos compiladores C++ irá requerer que você declare as funções-membro correspondentes fora da classe.

1083 OS MEMBROS DE CLASSE PODEM SER RECURSIVOS

Você aprendeu anteriormente na seção Funções que, uma função recursiva chama a si mesma para efetuar uma tarefa até que uma condição final específica seja encontrada. Quando você define funções de classe, as funções podem ser recursivas. O programa a seguir, *clas_str.cpp*, cria uma classe string com duas funções recursivas, *str_inversa* e *str_tamanho*:

```
#include <iostream.h>
#include <string.h>

class ClasseString  {
 public:
   void str_inverso(char *string)
   {
     if (*string)
       {
         str_inverso(string+1);
         cout.put(*string);
       }
   };
   int str_tamanho(char *string)
   {
       if (*string)
         return (1 + str_tamanho(++string));
       else
         return(0);
     };

   ClasseString(char *string)  { strcpy(ClasseString::string, string); };
   char string[256];
 };

void main(void)
 {
   ClasseString titulo("Bíblia do Programador C/C++");

   titulo.str_inverso(titulo.string);
   cout << endl << "O título tem "
        << titulo.str_tamanho(titulo.string)
        << " bytes de comprimento.";
 }
```

O programa *clas_str.cpp* primeiro cria uma ocorrência da classe *ClasseString* chamada *titulo*. O programa então inverte o título e mostra-o, e, depois, mostra uma string com o tamanho da string. Quando você compilar e executar o programa *clas_str.cpp*, sua tela exibirá a seguinte saída:

```
++C/C rodamargorP od ailbíB
O título tem 27 bytes de comprimento
C:\>
```

COMPREENDENDO O PONTEIRO THIS 1084

Toda vez que seu programa cria uma ocorrência de classe, C++ cria um ponteiro especial chamado *this*, que contém o endereço da ocorrência do objeto atual. C++ reconhece o ponteiro *this* somente quando um membro não-estático da ocorrência do objeto está executando. As ocorrências, por sua vez, usam o ponteiro *this* para acessar os diferentes métodos. No entanto, normalmente, o uso de *this* é transparente (em outras palavras, não é necessário). O compilador atribui *this* e efetua os redirecionamentos necessários automaticamente. Seus programas em geral não precisam usar o ponteiro *this*, mas podem e muitos programadores fazem isso por questões de clareza. O programa a seguir, *exibisto.cpp*, usa o ponteiro *this* para exibir os valores de vários membros de ocorrência. O programa também exibe os valores sem usar o ponteiro *this* para ilustrar que o compilador automaticamente insere instruções para efetuar a indireção correta para você:

```
#include <iostream.h>
#include <string.h>

class AlgumaClasse
 {
  public:
    void exibe_com_this(void)
    {
      cout << "Livro: " << this->titulo << endl;
      cout << "Autor: " << this->autor << endl;
    };

   void exibe_sem_this(void)
   {
     cout << "Livro: " << titulo << endl;
     cout << "Autor: " << autor << endl;
   };

   AlgumaClasse(char *titulo, char *autor)
   {
     strcpy(AlgumaClasse::titulo, titulo);
     strcpy(AlgumaClasse::autor, autor);
   };

  private:
    char titulo[256];
    char autor[256];
 };

void main(void)
 {
   AlgumaClasse livro("Bíblia do Programador C/C++", "Jamsa e Klander");

   livro.exibe_com_this();
   livro.exibe_sem_this();
 }
```

Quando você compilar e executar o programa *exibisto.cpp*, sua tela exibirá a seguinte saída:

```
Livro: Bíblia do Programador C/C++
Autor: Jamsa e Klander
Livro: Bíblia do Programador C/C++
Autor: Jamsa e Klander
C:\>
```

1085 COMO O PONTEIRO THIS DIFERE DOS OUTROS PONTEIROS

Na dica anterior você aprendeu que toda vez que seu programa chama o método de uma ocorrência, o compilador pré-atribui um ponteiro especial chamado *this* para apontar para a ocorrência do objeto. O ponteiro *this* é diferente dos outros ponteiros porque seu valor muda com diferentes invocações de ocorrência, de modo que seus programas precisam usar o ponteiro *this* com cuidado. À medida que você examinar programas C++, poderá encontrar comandos que retornam o valor para o qual *this* aponta, como mostrado aqui:

```
return(*this);
```

Em muitos casos, o compilador converterá o valor do ponteiro para uma referência, permitindo que um método retorne uma referência a uma ocorrência. Você precisa verificar o valor de retorno do membro atentamente para determinar se o método está retornando um ponteiro ou um valor de referência.

1086 COMPREENDENDO A AMARRAÇÃO PRECOCE OU TARDIA

Ao ler artigos e livros sobre a resolução das chamadas de função, você poderá encontrar termos como *amarração precoce* (*early binding*, no tempo da compilação) e *tardia* (*late binding*, em tempo de execução). Os termos descrevem quando o endereço para cada função que seus programas chamarão será resolvido (tornado conhecido para o programa). Neste ponto, o compilador resolveu todos os endereços de função-membro de classe que você usou na compilação ou na ligação. A resolução do endereço neste momento é chamada precoce (algumas vezes estática). C++ também suporta a *amarração dinâmica* por meio do uso de funções *virtuais*. A amarração tardia (também chamada de *dinâmica*) ocorre em tempo de execução, e dá aos programas que usam herança múltipla grande flexibilidade. Várias dicas a seguir discutem as funções virtuais em detalhes. As funções virtuais permitem que C++ suporte polimorfismo, que foi apresentada na seção Objetos.

1087 PONTEIROS PARA CLASSES

À medida que seus programas forem se tornando mais complexos, você poderá eventualmente trabalhar com ponteiros para objetos. Por exemplo, o programa a seguir, *ptr_obj.cpp*, cria uma única classe-base e uma única classe derivada. O programa usa o operador *new* para alocar dinamicamente ocorrências de cada tipo de classe e usa indireção de ponteiro para chamar os métodos de cada ocorrência:

```
#include <iostream.h>

class Base
 {
 public:
   void mensa_base(void)  { cout << "Esta é a classe-base\n"; };
 };

class Derivada: public Base
 {
 public:
   void mensa_derivada(void)  { cout << "Esta é a classe derivada\n" ; };
 };

void main(void)
 {
   Base *ponteiro_base = new Base;
   Derivada *ponteiro_derivado = new Derivada;

   ponteiro_base->mensa_base();
   ponteiro_derivado->mensa_derivada();
 }
```

Como você pode ver, acessar os elementos de cada classe usando um ponteiro e um operador de membro é fundamentalmente idêntico a acessar os membros de cada classe usando uma ocorrência de objeto e o operador de ponto. No entanto, como você aprenderá em dicas posteriores, se usar um ponteiro para acessar a classe-base,

poderá mais tarde usar o mesmo ponteiro para acessar as ocorrências da classe derivada. Alternativamente, seus programas podem usar um ponteiro para uma classe derivada para apontar para uma classe-base. A dica a seguir discute o uso de um único ponteiro com diferentes classes em detalhes.

USANDO O MESMO PONTEIRO PARA DIFERENTES CLASSES 1088

Na dica anterior, você criou dinamicamente ocorrências das classes *Base* e *Derivada*. Para fazer isso, o programa *ptr_obj.cpp* usou duas variáveis ponteiro diferentes — uma declarada como um ponteiro para o tipo *Base* e uma declarada como um ponteiro para o tipo *Derivada*. Felizmente, quando seus programas usam herança, C++ lhe permite usar um ponteiro para a classe-base para apontar para uma classe derivada. No entanto, ao usar o ponteiro de classe-base, você poderá somente acessar membros da classe-base original; não poderá acessar os membros da classe derivada. O programa a seguir, *baseptr.cpp*, atribui o ponteiro de classe-base para apontar para a classe derivada. O programa então usa o ponteiro para acessar o membro de classe-base *mensa_base*:

```
#include <iostream.h>

class Base
 {
  public:
    void mensa_base(void)
          { cout << "Esta é a classe-base\n"; };
 };

class Derivada: public Base
 {
  public:
    void mensa_derivada(void)  { cout << "Esta é a classe derivada\n" ; };
 };

void main(void)
 {
   Base *ponteiro_base = new Base;
   ponteiro_base->mensa_base();
   ponteiro_base = new Derivada;
   ponteiro_base->mensa_base();
 }
```

Como você aprenderá na dica a seguir, quando uma classe derivada e uma classe-base têm os mesmos nomes de membro, e você usa um ponteiro para a base para apontar para uma classe derivada, talvez não obtenha os resultados esperados.

CONFLITOS DE NOME DERIVADO E BASE COM PONTEIROS 1089

C++ lhe permite apontar para uma classe derivada usarem um ponteiro que declarou como um ponteiro para a classe-base. O programa a seguir, *nomebase.cpp*, usa um ponteiro de classe-base para apontar para uma classe derivada. As classes *base* e *derivada* têm o nome de membro *exibe_mensa*:

```
#include <iostream.h>

class Base
 {
  public:
    void exibe_mensa(void)  { cout << "Esta é a classe-base\n"; };
 };

class Derivada: public Base
 {
  public:
    void exibe_mensa(void) { cout << "Esta é a classe derivada\n" ; };
 };
```

```
void main(void)
 {
   Base *ponteiro_base = new Base;
   ponteiro_base->exibe_mensa();

   ponteiro_base = new Derivada;
   ponteiro_base->exibe_mensa();
 }
```

Quando você compilar e executar o programa *nomebase.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a classe-base
Esta é a classe-base
C:\>
```

Por padrão, quando as classes *base* e *derivada* usarem os mesmos nomes de função, e você usar um ponteiro para a classe-base, o compilador C++ resolverá o ponteiro para a função da classe-base. No entanto, algumas vezes, você desejará que o compilador chame um membro de classe derivada. Como verá na dica a seguir, você precisará usar funções virtuais para fazer isso.

1090 COMPREENDENDO AS FUNÇÕES VIRTUAIS

Você aprendeu que, quando uma classe herda os métodos de outra classe, algumas vezes os nomes dos membros da classe estarão em conflito. Se você estiver usando um ponteiro de classe-base para acessar uma classe derivada e chamar um dos membros com o mesmo nome que um membro de classe-base, o membro de classe-base executará. No entanto, se quiser que C++ chame o membro de classe derivada, você precisará definir uma *função virtual* para esse membro de classe-base. Usar funções virtuais não é muito diferente do que usar as operações que você já executou. Para criar uma função virtual, você simplesmente precederá o nome da função com a palavra-chave *virtual*. O tipo de retorno e a lista de parâmetros da função precisam ser idênticos para cada função virtual. O programa a seguir, *virt_um.cpp*, define a função *exibe_mensa* como uma função virtual dentro das classes *base* e *derivada*:

```
#include <iostream.h>

class Base
 {
  public:
    virtual void exibe_mensa(void) { cout << "Esta é a classe-base\n"; };
 };

class Derivada: public Base
 {
  public:
    virtual void exibe_mensa(void) { cout << "Esta é a classe derivada\n" ; };
 };

void main(void)
 {
   Base *ponteiro_base = new Base;
   ponteiro_base->exibe_mensa();

   ponteiro_base = new Derivada;
   ponteiro_base->exibe_mensa();
 }
```

Quando você compilar e executar o programa *virt_um.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a classe-base
Esta é a classe derivada
C:\>
```

Como você pode ver, como ambas as classes usam funções virtuais, o ponteiro pode chamar corretamente os métodos *Base* e *Derivada*.

HERDANDO O ATRIBUTO VIRTUAL 1091

Quando as classes dentro de seus programas herdam uma função virtual, a função herdada mantém a natureza virtual da função-base. Em outras palavras, independentemente de quantas vezes seu programa herdar uma função virtual, ela permanecerá virtual. Por exemplo, quando uma classe derivada herda uma função virtual e depois seu programa usa a classe derivada como uma classe-base, a classe derivada removida duas vezes ainda pode anular a função virtual. Para compreender melhor como o membro dentro da classe derivada pode anular a função virtual, considere o seguinte programa, *funcv_in.cpp*:

```
#include <iostream.h>

class base
 {
 public:
   virtual void func_v(void) { cout << "Esta é a func_v() de base." << endl;
 };

class derivada1 : public base
 {
  public:
    void func_v(void) { cout << "Esta é func_v() de derivada1." << endl; };

class derivada2 : public derivada1
 {
  public:
    void func_v(void)
     { cout << "Esta é a func_v() de derivada2." << endl;  }
 };

void main(void)
 {
   base *p, b;
   derivada1 d1;
   derivada2 d2;

   p = &b;     // Aponta para a classe-base
   p->func_v();
   p = &d1;    // Aponta para a primeira classe derivada
   p->func_v();
   p = &d2;    // Aponta para a segunda classe derivada
   p->func_v();
 }
```

Em ambos os casos (*derivada1* e *derivada2*), as definições na classe de *func_v* anulam a definição virtual dentro de *base*. Portanto, quando o programa *funcv_in.cpp* muda o ponteiro para apontar para as classes derivadas em sequência, o acesso do membro *func_v* executa os *func_vs* locais que cada classe define. Quando você compilar e executar o programa *funcv_in.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a func_v() de base.
Esta é a func_v() de derivada1.
Esta é a func_v() de derivada2.
C:\>
```

1092 AS FUNÇÕES VIRTUAIS SÃO HIERÁRQUICAS

Como você aprendeu nas dicas anteriores, a herança é hierárquica. Portanto, as funções virtuais também precisam ser hierárquicas. Assim, se uma classe derivada não anular a função virtual, o compilador usará a versão derivada mais próxima acima dessa classe na árvore. No programa a seguir, por exemplo, *derivada2* é derivada de *derivada1*, que, por sua vez, é derivada de *base*. No entanto, *derivada2* não anula a função *func_v*, de modo que o compilador, em vez disso, usa a *func_v* anulada dentro da definição da classe *derivada1*. À medida que trabalhar com as funções virtuais, você poderá querer testar sua hierarquia de herança usando um programa como o *hier_vir.cpp*, como mostrado aqui:

```
#include <iostream.h>

class base
 {
  public:
    virtual void func_v(void) { cout << "Esta é func_v() de base." << endl;  }
 };

class derivada1 : public base
 {
  public:
    void func_v(void) { cout << "Esta é func_v() de derivada." << endl; }
 };

class derivada2 : public derivada1 { };

void main(void)
 {
   base *p, b;
   derivada1 d1;
   derivada2 d2;

   p = &b;      // Aponta para a classe-base
   p->func_v();
   p = &d1;     // Aponta para a primeira classe derivada
   p->func_v();
   p = &d2;     // Aponta para a segunda classe derivada
   p->func_v();   // No entanto, usa a função func_v de  derivada1
 }
```

No caso do programa *hier_vir.cpp*, a mudança do ponteiro para a classe derivada não causa impacto em qual função o operador-membro chama, porque a classe derivada não define sua própria implementação do membro *func_v*. Em vez disso, o último acesso chama o membro *func_v* para a classe *derivada1*. Quando você compilar e executar o programa *hier_vir.cpp*, sua tela exibirá a seguinte saída:

```
Esta é a func_v() de base.
Esta é a func_v() de derivada1.
Esta é a func_v() de derivada2.
C:\>
```

1093 IMPLEMENTANDO O POLIMORFISMO

Na seção Objetos vimos que o polimorfismo é a capacidade do mesmo objeto assumir formas diferentes. C++ suporta polimorfismo usando funções virtuais. Usando funções virtuais, o mesmo ponteiro pode apontar para classes diferentes para efetuar diferentes operações. O programa a seguir, *polimorf.cpp*, cria uma classe-base e duas classes derivadas. O programa *polimorf.cpp* então usa o ponteiro *pols* para chamar diferentes métodos:

```
#include <iostream.h>
#include <stdlib.h>
class Base
```

```
{
 public:
   virtual int soma(int a, int b) { return(a + b); };
   virtual int sub(int a, int b)  { return(a - b); };
   virtual int mult(int a, int b) { return(a * b); };
};

class ExibeMat : public Base
{
  virtual int mult(int a, int b)
   {
     cout << a * b << endl;
     return(a * b);
   };
};

class SubtPositiva : public Base
{
  virtual int sub(int a, int b) { return(abs(a - b)); };
};

void main(void)
{
  Base *poli = new ExibeMat;

  cout << poli->soma(562, 531) << ' ' << poli->sub(1500, 407) << endl;
  poli->mult(1093, 1);

  poli = new SubtPositiva;
  cout << poli->soma(892, 201) <<' '  << poli->sub(0, 1093)  << endl;
  cout << poli->mult(1, 1093);
}
```

O programa *polimorf.cpp* mostra claramente como o polimorfismo e as funções virtuais expandem o poder da herança para permitir que seus programas obtenham resultados interessantes. Observe que, dependendo da ocorrência apontada pelo ponteiro *poli*, as operações que *poli* efetua podem diferir. No entanto, quando você compilar e executar o programa *polimorf.cpp*, sua tela exibirá a seguinte saída:

```
1093  1093
1093
1093  1093
1093
C:\>
```

COMPREENDENDO AS FUNÇÕES VIRTUAIS PURAS 1094

Quando você deriva uma classe a partir de outra, C++ lhe permite usar funções virtuais para controlar quais funções da classe o programa chama quando você usa um ponteiro de classe-base para apontar para uma classe derivada. Ao ler mais sobre funções virtuais, você poderá encontrar o termo *função virtual pura*. Uma função virtual pura é similar a um protótipo que você declara na classe-base para a qual a classe-base requer a classe derivada para fornecer uma implementação. Dentro da classe-base, uma função virtual pura aparece, como a seguir:

```
virtual tipo nome_função(parâmetros) = 0;
```

O sinal de igual (=) e o valor zero (0) que seguem o protótipo indicam que a função é uma função virtual pura para a qual o programa precisa fornecer uma implementação. O programa a seguir, *virtpura.cpp*, ilustra uma função virtual pura:

```
#include <iostream.h>
#include <string.h>
```

```
class Base
 {
  public:
    virtual void exibe_mensa(void)  { cout << "Mensagem da classe-base" <<
     endl; };
    virtual void exibe_inverso(void) = 0;
 };

class Derivada : public Base
 {
  public:
    virtual void exibe_mensa(void) { cout << "Mensagem da classe derivada"
     << endl; };
    virtual void exibe_inverso(void)
      { cout << strrev("Mensagem da classe derivada") << endl; };
 };

void main(void)
 {
   Base *poli = new Derivada;
   poli->exibe_mensa();
   poli->exibe_inverso();
 }
```

É importante que você observe que uma função virtual pura *requer* que cada classe derivada defina sua própria implementação da função, enquanto uma função virtual permite que as classes derivadas usem a função que a classe-base deriva. Por exemplo, no programa *virtpura.cpp*, se *poli* apontasse para base, seu programa não poderia ter acesso ao membro *exibe_inverso*. O ponteiro *poli* somente pode acessar *exibe_inverso* na classe *derivada*.

1095 COMPREENDENDO AS CLASSES ABSTRATAS

Na dica anterior você aprendeu que uma função virtual pura é um protótipo de função para a qual a classe-base requer que uma classe derivada forneça uma implementação. Quando uma classe contém somente funções virtuais puras, C++ se refere a essa classe como uma *classe abstrata*. Em geral, uma classe abstrata fornece um gabarito a partir do qual seus programas podem mais tarde derivar outras classes. C++ não lhe permite criar uma variável de um tipo de classe abstrato. Se você tentar fazer isso, o compilador gerará um erro de sintaxe.

As classes abstratas também são conhecidas como *fábricas de classe*, pois seus programas as utilizam como locais centrais a partir dos quais derivam (ou fabricam, para manter a analogia) outras classes dentro de seu programa.

1096 USANDO FUNÇÕES VIRTUAIS

Um aspecto central da programação orientada a objetos é "uma interface, múltiplos métodos". Em outras palavras, você pode criar classes-base em C++ que seus programas usarão para definir a natureza de uma interface para uma classe geral. Cada classe que você, então, deriva a partir de uma classe-base implementa as operações específicas que elas relacionam ao tipo de dados que o tipo derivado usa.

Um dos modos mais poderosos de seus programas alcançarem o objetivo "uma interface, múltiplos métodos" é usar funções virtuais, classes abstratas e polimorfismo em tempo de execução. Os programas que usam todos esses recursos lhe permitem criar uma hierarquia de classe, que vai do geral para o complexo (base para derivada). Você criará todos os aspectos comuns e interfaces dentro de uma classe-base. Nos casos em que você pode implementar certas ações somente dentro de uma classe derivada, a classe-base deve definir uma função virtual para criar a interface que a classe derivada usará. Para compreender melhor o conceito "uma interface, múltiplos métodos", considere o programa a seguir, *funcv_ex.cpp*, que aplica o conceito "uma interface, múltiplos métodos" para algumas classes simples:

```
#include <iostream.h>

class converte
 {
 protected:
```

```
    double val1;
    double val2;
  public:
    converte(double i) { val1 = i; }
     double obtem_conv(void) {return val2;}
     double obtem_inic(void) {return val1;}
     virtual void calcula(void) = 0;
  };

// litros para galoes
class l_para_g : public converte
  {
   public:
    l_para_g(double i) : converte(i) { }
    void calcula(void) { val2 = val1 / 3.7854; }
  };

// Fahrenheit para Centigrados
class f_para_c : public converte
  {
  public:
    f_para_c(double i) : converte(i) { }
    void calcula(void) { val2 = (val1 - 32) / 1.8; }
  };

void main(void)
  {
    converte *p;     // ponteiro para a classe-base

    l_para_g lgob(4);
    f_para_c fcob(70);

    p = &lgob;      // converte litros para galoes
    cout << p->obtem_inic() << " litros são ";
    p->calcula();
    cout << p->obtem_conv() << " galões." << endl;
    p = &fcob;     // converte fahrenheit para centigrados
    cout << p->obtem_inic() << " em Fahrenheit são ";
    p->calcula();
    cout << p->obtem_conv() << " Centigrados." << endl;
  }
```

O programa *funcv_ex.cpp* deriva as classes *l_para_g* (litros para galões) e *f_para_c* (Fahrenheit para Centígrados) a partir da classe-base *Converte*. Ambas as classes derivadas inicializam o membro *i* dentro da classe *Converte*. Ambas as funções também anulam a função virtual pura *calcula*. No entanto, ambas as classes derivadas usam a função *obtem_conv* e *obtem_inic*, que a classe-base define. E ambas as classes derivadas usam os membros *val1* e *val2* da classe-base. Na verdade, a única função que difere entre as duas classes derivadas é o membro *calcula*, que usa diferentes cálculos para cada classe derivada. Quando você compilar e executar o programa *funcv_ex.cpp*, sua tela exibirá a seguinte saída:

```
4 litros são 1.05669 galões.
70 em Fahrenheit são 21.111 Centígrados.
C:\>
```

MAIS SOBRE AMARRAÇÃO PRECOCE OU TARDIA 1097

Como você aprendeu, ao discutir programas C++ com outros programadores, freqüentemente você ouvirá os termos *amarração precoce* e *amarração tardia*. A amarração precoce se refere aos eventos que ocorrem no momento da compilação. Em outras palavras, o compilador conhece todas as informações necessárias para compilar o pro-

grama. Exemplos típicos de amarração precoce incluem as chamadas de funções normais (tais como as chamadas de biblioteca padrão), chamadas de função sobrecarregadas e operadores sobrecarregados.

Por outro lado, a amarração tardia se refere aos eventos que seu programa só resolve em tempo de execução. Para seus programas usarem a amarração tardia, as classes dentro desses programas precisam declarar funções virtuais. Como você sabe, ao acessar uma função virtual por meio de um ponteiro-base, o programa determina qual função virtual chamar com base no objeto real que o ponteiro-base referencia. Como o compilador não tem um modo de saber qual objeto o ponteiro referencia no momento da compilação, o programa precisa responder às referências de ponteiro em tempo de execução.

1098 DECIDINDO ENTRE A AMARRAÇÃO PRECOCE E TARDIA

Na dica anterior vimos que, seus programas geralmente incluirão amarração precoce, amarração tardia, ou ambas. À medida que você escrever programas mais complexos, verá que usará a amarração tardia mais freqüentemente com seus programas mais complexos do que com os programas mais simples. À medida que você for acrescentando mais funções virtuais aos seus programas, começará a decidir entre amarração precoce e tardia para seus objetos com mais freqüência.

A principal vantagem da amarração precoce é a eficiência. Como o computador pode resolver a função inteira bem antes da execução do programa, as funções de amarração precoce são muito eficientes. Além disso, a amarração precoce é menos sujeita a erros na execução porque o compilador pode detectar muitos problemas com antecedência. Por outro lado, a principal vantagem da amarração tardia é a flexibilidade. Ao contrário da amarração precoce, a tardia permite que seus programas respondam aos eventos que ocorrem durante a execução do programa sem criar uma grande quantidade de "código de contingência".

1099 UM EXEMPLO DE AMARRAÇÃO PRECOCE E AMARRAÇÃO TARDIA

Em dicas anteriores, você criou classes que usaram funções virtuais e classes que não usaram. Como você aprendeu nas Dicas 1097 e 1098, os programadores descrevem programas que usam funções virtuais como programas de amarração tardia. Os programas que não usam funções virtuais são programas de amarração precoce. Por exemplo, o fragmento de código a seguir mostra como você poderia declarar a classe *converte* definida na Dica 1096 com uma função virtual:

```
class converte
 {
  protected:
    double val1;
    double val2;
  public:
    converte(double i)  { val1 = i; }
    double obtem_conv(void)  { return val2; }
    double obtem_inic(void)  { return val1; }
    virtual void  calcula(void) = 0;
 };

// litros para galoes
class l_para_g : public converte
  {
    public :
      l_para_g(double i) : converte(i)  { }
      void calcula(void)   {  val2 = val1 / 3.7854;  }
 };
```

Alternativamente, seu programa poderia declarar *l_para_g* com amarração precoce, como mostrado aqui.

```
class l_para_g
 {
   protected:
     double val1;
```

```
    double val2;
  public:
    l_para_g(double i)  { val1 = i; }
    double obtem_conv(void)  { return val2; }
    double obtem_inic(void)  { return val1; }
    void  calcula(void)  { val2 = val1 / 3.7854; };
};
```

Como você pode ver, no fragmento de código precedente, a amarração tardia não é particularmente útil. No entanto, se você fosse acrescentar outras conversões ao programa, eventualmente criaria muitas classes repetitivas que compartilhariam processamento repetitivo. Assim como você provavelmente usaria herança para solucionar o problema da classe repetitiva, deveria usar a amarração tardia para tornar seu programa mais eficiente e mais claro para o leitor.

DEFININDO UM MANIPULADOR DE CANAL DE SAÍDA 1100

Várias dicas na seção Introdução à Linguagem C++ usaram manipuladores de canal de saída, tais como *hex* e *endl*. Como você aprendeu anteriormente, pode criar seus próprios manipuladores de canal de saída. Por exemplo, você poderia querer escrever um programa que criasse um novo manipulador chamado *atencao*, que soasse o alto-falante interno do computador para chamar a atenção do usuário. Você poderia então usar o manipulador *atencao* dentro do canal *cout*, como mostrado aqui:

```
cout << atencao << "Acho que seu disco está com defeito";
```

O código a seguir implementa *manipul.cpp*, que cria o manipulador *atencao*:

```
#include <iostream.h>

ostream& atencao(ostream& cout) { return(cout << '\a'); };

void main(void)
 {
  cout << atencao << "O chefe está vindo ao escritório...\n";
 }
```

É HORA DE DAR UMA OLHADA EM IOSTREAM.H 1101

Na seção Introdução à Linguagem C++, você foi advertido a não examinar o arquivo de cabeçalho *iostream.h*. Agora que você domina as classes, sobrecarga, funções virtuais e os fundamentos sobre a programação orientada a objetos, deverá não somente examinar o arquivo, como também examinar cada linha dele. Os desenvolvedores do compilador C++ usaram várias técnicas interessantes de programação ao criar *iostream.h* — técnicas que você também pode usar em seus programas. Inicialmente, imprima a listagem do arquivo para poder referenciá-lo quando criar suas próprias classes. Segundo, crie uma cópia do arquivo e a nomeie como *iostream.nts*. Em seguida, leia parte do arquivo todo dia e acrescente comentários que expliquem o processamento do programa. Se você examinar duas páginas por dia, completará o arquivo inteiro em menos de uma semana. Você não apenas se tornará um especialista nas operações de C++, mas aprenderá muito sobre as definições de classes complexas.

USANDO SIZEOF COM UMA CLASSE 1102

Como você sabe, o operador *sizeof* retorna o número de bytes necessários para armazenar um objeto. À medida que seus programas trabalharem com objetos, algumas vezes você precisará conhecer o tamanho do objeto. Por exemplo, assuma que você esteja lendo um arquivo de objetos. Você pode usar o operador *sizeof* para determinar o tamanho de cada objeto dentro do arquivo. O operador *sizeof* retorna somente o tamanho dos dados-membro da classe. O programa a seguir, *sizeof.cpp*, usa o operador *sizeof* para determinar o tamanho de duas classes. A primeira classe é uma classe-base e a segunda é uma classe derivada:

```
#include <iostream.h>
#include <string.h>
```

```
class Base {
 public:
   Base(char *mensagem) { strcpy(Base::mensagem, mensagem); };
   void exibe_base(void) { cout << mensagem << endl; };
 private:
   char mensagem[256];
 };

class Derivada: public Base {
 public:
   Derivada(char *dmsg, char *bmsg) : Base(bmsg)  {
      strcpy(mensagem, dmsg); };
   void exibe_derivada(void)
   {
     cout << mensagem << endl;
     exibe_base();
   };
 private:
   char mensagem[256];
 };

void main(void)
 {
   Base alguma_base("Isto é uma base");
   Derivada alguma_derivada("Mensagem derivada", "Mensagem base ");

   cout << "O tamanho da classe-base é " << sizeof(alguma_base) << " bytes"
    << endl;
   cout << "O tamanho da classe derivada é " << sizeof(alguma_derivada) <<
    " bytes" << endl;
 }
```

Quando você compilar e executar o programa *sizeof.cpp*, sua tela exibirá a seguinte saída:

```
O tamanho da classe-base é 256 bytes
O tamanho da classe derivada é 512 bytes
```

1103 As Palavras-chave Private, Public e Protected Também Podem Ser Aplicadas às Estruturas

Várias dicas nesta seção usaram membros de classe privados, públicos e protegidos. Quando seus programas C++ usam estruturas, você também pode ter membros privados, públicos e protegidos. Por padrão, todos os membros de estruturas são públicos. No entanto, você pode usar os rótulos *private* e *protected* para identificar membros para os quais quer controlar o acesso. O programa a seguir, *privst.cpp*, ilustra o uso de membros privados dentro de uma estrutura. Como você verá, a maioria das capacidades que C++ fornece para as classes também existe para as estruturas:

```
#include <iostream.h>
#include <string.h>

struct MeuLivro  {
  char titulo[64];   // Público por padrão
  void exibe_livro(void)
    {
      cout << "Livro: " << titulo << " Preço: $"
           << preco;
    };

  void def_preco(float amount) { preco = amount; };
  void atribui_titulo(char *nome) { strcpy(titulo, nome); };
```

```
  private:
    float preco;
 };

void main(void)
 {
   MeuLivro livro;

   livro.atribui_titulo("Bíblia do Programador C/C++");
   livro.def_preco(49.95);
   livro.exibe_livro();
 }
```

Quando você compilar e executar o programa *privest.cpp*, ele construirá o objeto *livro*, atribuirá o título e o preço e depois exibirá as informações de preço e título do objeto *livro* na sua tela, como mostrado aqui:

```
Livro: Bíblia do Programador C/C++    Preço: $49.95
C:\>
```

No programa *privest.cpp*, a estrutura *MeuLivro* especifica o membro *preco* como privado. Portanto, o único modo de acessar o membro é usar um dos métodos públicos da estrutura. Caso ache que suas estruturas requerem esse tipo de proteção de membro, você deve começar a trabalhar com classes.

COMPREENDENDO AS CONVERSÕES DE CLASSES 1104

Como sabe, quando você passa um valor do tipo *int* a uma função que requer um valor *long*, C++ promove o valor *int* para tipo correto. Da mesma forma, se você passa um parâmetro do tipo *float* a uma função que requer um valor *double*, C++ faz uma conversão similar. Quando você trabalha com classes de C++, também pode especificar as conversões que C++ deve efetuar para converter valores de classe para um tipo de dado padrão (tal como *int* ou *long*) ou até para uma classe diferente. As conversões são comuns quando você passa um parâmetro para um tipo para uma função construtora para uma classe de um tipo diferente. Por exemplo, assuma que a classe *DadosLivro* receba três strings de caracteres como parâmetros para sua construtora:

```
DadosLivro (char *titulo, char *autor, char *editora)
 {
   // Comandos
 }
```

No entanto, assuma que o programa periodicamente chame a construtora com uma estrutura do tipo *InfoLivro*, como mostrado aqui:

```
struct InfoLivro
 {
   char titulo[64];
   char autor[64];
   char editora[64];
   float preco;
   int paginas;
 };
```

A classe pode criar uma segunda construtora que converta os dados apropriadamente. A dica a seguir ilustra um programa que usa uma estrutura e uma segunda construtora.

CONVERTENDO DADOS EM UMA CONSTRUTORA 1105

Como você aprendeu, algumas vezes precisa converter dados de um formato para um formato que a classe espera. Um modo fácil de efetuar tal conversão é usar diferentes funções construtoras. O programa a seguir, *converte.cpp*, usa tal função construtora para converter as informações que uma estrutura do tipo *LivroInfo* contém.

```
#include <iostream.h>
#include <string.h>
struct LivroInfo {
```

```
  char titulo[64];
  char editora[64];
  char autor[64];
  float preco;
  int paginas;
 };

class DadosLivro {
  public:
    DadosLivro(char *titulo, char *editora, char *autor);
    DadosLivro(struct LivroInfo);
    void exibe_livro(void)
     {
       cout << "Livro: " << titulo << " de " << autor << endl <<
         "Editora: " << editora
              << endl;
     };
  private:
    char titulo[64];
    char autor[64];
    char editora[64];
 };

DadosLivro::DadosLivro(char *titulo, char *editora, char *autor)
 {
   strcpy(DadosLivro::titulo, titulo);
   strcpy(DadosLivro::editora, editora);
   strcpy(DadosLivro::autor, autor);
 }

DadosLivro::DadosLivro(LivroInfo livro)
 {
   strcpy(DadosLivro::titulo, livro.titulo);
   strcpy(DadosLivro::editora, livro.editora);
   strcpy(DadosLivro::autor, livro.autor);
 }

void main(void)
 {
   LivroInfo livro = {"Aprendendo C++, Terceira Edição",
                      "Makron Books", "Jamsa", 29.95, 256 };

   DadosLivro grande_livro("Bíblia do Programador C/C++",
                           "Makron Books", "Jamsa e Klander");
   DadosLivro pequeno_livro(livro);

   grande_livro.exibe_livro();
   pequeno_livro.exibe_livro();
 }
```

O programa *converte.exe* começa seu processamento gerando uma ocorrência da classe *LivroInfo* no objeto *livro*. A declaração chama a construtora *LivroInfo*, que atribui os parâmetros às variáveis-membro do objeto *livro*. O programa então efetua processamento similar com o objeto *grande_livro*. A terceira declaração, *pequeno_livro(livro)*, chama a construtora sobrecarregada para a classe *DadosLivro*, que então tira seus valores necessários de dentro do objeto livro e atribui os valores ao novo objeto *pequeno_livro*. Quando você compilar e executar o programa *converte.exe*, sua tela exibirá a seguinte saída:

```
Livro: Bíblia do Programador C/C++ de Jamsa e Klander
Editora: Makron Books
Livro: Aprendendo C++ Terceira Edição de Jamsa
Editora: Makron Books
C:\>
```

Usar a função construtora para efetuar a conversão, como mostrado nesta dica, realmente não é diferente da sobrecarga de operações que você executou em todo este livro.

ATRIBUINDO UMA CLASSE A OUTRA 1106

Na dica anterior, você criou uma função construtora que converteu os dados em uma estrutura do tipo *LivroInfo* nos campos de dados para uma ocorrência da classe *DadosLivro*. À medida que você for examinando programas C++, poderá encontrar comandos em que o programa atribui um tipo de classe para outro, como mostrado aqui:

```
class DadosLivro grande_livro;
char titulo[256];

titulo = grande_livro;
```

Neste caso, o programa atribui a classe *grande_livro* a uma variável string. Para suportar essas operações, seu programa precisa dizer ao compilador a conversão correta a aplicar. Para fazer isso, você precisa criar uma função-membro dentro da classe *DadosLivro* que efetua a conversão. Neste caso, a função atribuirá o título do livro à string. Existem duas regras que a função de conversão precisa seguir. Primeira, dentro da classe, o programa precisa definir a função como uma função de sobrecarga de operador, como a seguir:

```
operator char *(void);
```

Segunda: o código da função correspondente precisa retornar um valor do tipo convertido que, neste caso, é um ponteiro a uma string de caracteres. O programa a seguir, *atr_clas.cpp*, usa uma função-membro de conversão para atribuir a classe a uma string de caracteres. Neste caso, a função de conversão atribui o título do livro à string:

```
#include <iostream.h>
#include <string.h>

class DadosLivro {
  public:
    DadosLivro(char *titulo, char *editora, char *autor);
    void exibe_livro(void)
    {
       cout << "Livro: " << titulo << " de "
            << autor << " Editora: " << editora << endl;
    };
    operator char *();

  private:
    char titulo[64];
    char autor[64];
    char editora[64];
};

DadosLivro::DadosLivro(char *titulo, char *editora, char *autor)
 {
   strcpy(DadosLivro::titulo, titulo);
   strcpy(DadosLivro::editora, editora);
   strcpy(DadosLivro::autor, autor);
 }

DadosLivro::operator char *(void)
 {
   char *ptr = new char[256];

   return(strcpy(ptr, titulo));
 }

void main(void)
```

```
{
  DadosLivro grande_livro("Bíblia do Programador C/C++",
            "Makron Books", "Jamsa e Klander");

  char *titulo;
  titulo = grande_livro;
  cout << "O título do livro é " << titulo << endl;
}
```

Usando funções de conversão, tais como *atr_clas.cpp*, seus programas poderão converter de uma classe para outra, conforme for necessário. Quando você compilar e executar o programa *atr_clas.cpp*, sua tela exibirá a seguinte saída:

```
O título do livro é Bíblia do Programador C/C++.
C:\>
```

1107 USE AMIGAS PARA A CONVERSÃO

Quando você efetua conversões entre uma classe e outra, algumas vezes precisa acessar os membros privados de outra classe. Como você aprendeu, é possível especificar uma classe *amiga* ou uma função *amiga* para permitir que outra classe acesse os dados privados sem tornar a classe visível para outras partes do seu programa. Se uma classe precisar acessar os dados privados de outra classe para efetuar uma conversão, especifique a função de conversão como uma *amiga* da classe, como mostrado aqui:

```
friend void Converte(DadosLivro &novo, InfoLivro livro);
// Definições da classe
void Converte(DadosLivro, &novo, InfoLivro livro)
 {
   strcpy(novo.titulo, livro.titulo);
   strcpy(novo.editora, livro.editora);
   strcpy(novo.autor, livro.autor) ;
 }
```

1108 DETERMINANDO QUANDO OS OPERADORES AUMENTAM OU REDUZEM A LEGIBILIDADE

Você aprendeu que, quando define uma classe, C++ lhe permite sobrecarregar um ou mais operadores. No entanto, antes de sobrecarregar um operador, você precisará determinar se a sobrecarga tornará o programa mais fácil ou mais difícil de compreender. Por exemplo, o programa a seguir, *str_mais.cpp*, sobrecarrega o operador de adição para a classe *String*. O programa depois usa o operador, bem como a função *anexastr*, para anexar o conteúdo de uma string em outra. Examine o programa e determine qual técnica é mais compreensível:

```
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

class String {
  public:
    char *operator +(char *anexa_str)
      { return(strcat(buffer, anexa_str)); };

    String(char *string)
     {
       strcpy(buffer, string);
       tamanho = strlen(buffer);
     }

    void exibe_string(void) { cout << buffer; };
    void anexastr(char *fonte) { strcat(buffer, fonte); };
```

```
 private:
   char buffer[256];
   int tamanho;
 };

void main(void)
 {
  String titulo("Bíblia do Programador ");
  titulo = titulo + "C++ do Jamsa\n";
  titulo.exibe_string();

  String livro2("Aprendendo C++");
  livro2.anexastr(", Terceira Edição ");
  livro2.exibe_string();
 }
```

Muitos programadores novos em C++ sobrecarregam mais operadores do que o necessário. A decisão que você precisa tomar ao criar seus programas é se a sobrecarga aumenta ou não a legibilidade do seu programa.

COMPREENDENDO OS GABARITOS 1109

Provavelmente muitas vezes seus programas precisam duplicar uma função para que ela suporte parâmetros de um tipo diferente. Por exemplo, a seguinte função, *compara_valores*, compara dois valores do tipo *int* e retorna o maior valor:

```
int compara_valores(int a, int b)
 {
   return ((a > b) ? a: b);
 }
```

Se, mais tarde, seu programa precisar comparar dois valores de ponto flutuante, você precisará criar uma segunda função. A segunda função é idêntica em processamento mas suporta diferentes tipos, como a seguir:

```
float compara_valores(float a, float b)
 {
   return ((a > b) ? a: b);
 }
```

Se as duas funções aparecem no mesmo programa, você precisa sobrecarregar a função original (criando a possibilidade de confusão) ou selecionar um nome exclusivo para cada função. Para lhe ajudar a reduzir essas definições de função duplicada e tornar seus programas mais fáceis de compreender, C++ suporta a definição de *gabaritos*. Um gabarito fornece os formatos das funções e marcadores de lugar de tipo. O código a seguir mostra a forma geral de um gabarito de função, em que *T* é um tipo que o compilador substituirá mais tarde:

```
template<class T> T nome_função(T param_a, T param_b)
 {
   // Comandos
 }
```

Por exemplo, considere o seguinte gabarito de função para *compara_valores:*

```
template<class T> T compara_valores(T a, T b)
 {
   return((a > b) ? a: b);
 }
```

Como você pode ver, o compilador pode substituir a letra *T* com o tipo *float* ou *int* para criar as funções mostradas anteriormente. Várias dicas a seguir discutem os gabaritos em detalhe.

Nota: O compilador Turbo C++ Lite que vem no CD-ROM que acompanha este livro não suporta definições genéricas. Para escrever programas que usam definições genéricas, você precisa usar outro compilador, tal como o Borland C++ 5.02 ou o Microsoft Visual C++.

1110 COLOCANDO UM GABARITO SIMPLES EM USO

C++ suporta o uso de funções de gabarito, como você viu na dica anterior. O programa a seguir, *compare.cpp*, usa o gabarito de função *compare_valores* para comparar valores de tipos diferentes:

```
#include <iostream.h>

template<class T> T compare_valores(T a, T b)
 {
   return((a > b) ? a: b);
 }

float compare_valores(float a, float b);
int compare_valores(int a, int b);
long compare_valores(long a, long b);

void main(void)
 {
   float a = 1.2345, b = 2.34567;
   cout << "Comparando " << a << ' ' << b  << ' ' << compare_valores(a, b)
<< endl;

   int c = 1, d = 1001;
   cout << "Comparando " << c << ' ' << d  << ' ' << compare_valores(c, d)
<< endl;

   long e = 1010101L, f = 2020202L;
   cout << "Comparando " << e << ' ' << f  << ' ' << compare_valores(e, f)
<< endl;
 }
```

No exemplo anterior, o gabarito no início do programa *compare cpp* especifica os comandos da função e os marcadores de lugar do tipo. Quando encontra os protótipos que aparecem antes de *main*, C++ cria as funções necessárias. Mais tarde, o compilador determinará qual função ou funções usar, com base no tipo dos parâmetros que o programa passará para o gabarito genérico (*float*, *int* ou *long*).

1111 COMPREENDENDO MELHOR AS FUNÇÕES GENÉRICAS

Você aprendeu que C++ suporta funções genéricas. Uma função genérica define um conjunto geral de operações que a função aplicará a vários tipos de dados. Uma função genérica recebe o tipo de dados no qual a função operará como um parâmetro. Como a função genérica é, por natureza, não explicitamente tipada, você pode usar o mesmo procedimento geral dentro da função em uma ampla variedade de tipos de dados. Criar funções genéricas dentro de seus programas pode ser útil porque muitos algoritmos são fundamentalmente iguais em seu processamento, porém dependentes do tipo de dado em que operam. Em dicas posteriores você aprenderá sobre a Biblioteca de Gabaritos Padrões (STL, de Standard Template Library), que aplica o conceito de funções genéricas a uma variedade de algoritmos generalizados. Por exemplo, como você aprendeu em dicas anteriores, o algoritmo quicksort é sempre o mesmo, independentemente do tipo de dado que o programa está classificando. Como o arquivo de cabeçalho criou o algoritmo quicksort como uma função genérica, você pode eliminar a necessidade de criar múltiplas funções que efetuam bas.camente o mesmo processamento.

Dentro de seus programas, você usará a palavra-chave *template* para criar funções genéricas. O significado da palavra "gabarito" é exata para o propósito das funções genéricas — em outras palavras, você usará funções genéricas para criar um gabarito de processamento que seus programas usarão com informações específicas. A forma geral da declaração de função genérica é mostrada aqui:

```
template <class Ttipo> tipo-retorno nome-função([lista parâmetros])
 {
   return corpo da função
 }
```

Ttipo é um nome marcador de lugar para o tipo de dados que a função usará. Você também pode usar o nome *Ttipo* dentro da definição da função. Em outras palavras, *Ttipo* representa um marcador de lugar que o compilador substituirá automaticamente com o tipo de dados correto cada vez que criar uma versão específica da função. Por exemplo, no programa *compare.cpp*, detalhado na dica anterior, o compilador substituirá a função genérica com três funções específicas: uma que retorna um valor *float*, uma que retorna um valor *int* e uma que retorna um valor *long*. Na dica a seguir, você aprenderá sobre os gabaritos que suportam múltiplos tipos.

GABARITOS QUE SUPORTAM MÚLTIPLOS TIPOS 1112

Quando você trabalha com gabaritos de C++, algumas vezes o gabarito requer mais de um tipo de dados. Por exemplo, considere a seguinte função, *soma_valores*, que soma um valor do tipo *long* e *int* e retorna um resultado *long*:

```
long soma_valores(long a, int b)
 {
   return (a + b);
 }
```

Para criar um gabarito para a função *soma_valores*, você precisa especificar dois tipos (como *T* e *T1*):

```
template<class T, class T1> T soma_valores(T a, T1 b)
 {
   return (a + b);
 }
```

No exemplo a seguir, o compilador substituirá cada ocorrência da classe *T* com o tipo que você especificar. Da mesma forma, o compilador substituirá cada ocorrência da classe *T1* com a classe correspondente que sua invocação especificar. O programa a seguir, *gabarito.cpp*, usa o gabarito *soma_valores*.

```
#include <iostream.h>

template<class T, class T1> T soma_valores(T a, T1 b)
  {
    return(a + b);
  }

long soma_valores(long a, int b);
double soma_valores(double a, float b);

void main(void)
 {
   long a = 320000L;
   int b = 31000;

   double c = 22.0 / 7.0;
   float d = 3.145;

   cout << "Somando " << a << ' ' << b << ' ' << soma_valores(a, b) << endl;
   cout << "Somando " << c << ' ' << d << ' ' << soma_valores(c, d) << endl;
 }
```

Quando você compilar e executar o programa *gabarito.cpp*, sua tela exibirá a seguinte saída:

```
Somando 320000 31000 351000
Somando 3.14286 3.145 6.2886
C:\>
```

1113 MAIS SOBRE GABARITOS COM MÚLTIPLOS TIPOS GENÉRICOS

Você pode declarar gabaritos que suportam múltiplos tipos. Por exemplo, na dica anterior, você criou uma função que suportava dois tipos e retornava o resultado como o primeiro tipo. No entanto, suas funções genéricas podem suportar um número ilimitado de tipos genéricos. A forma da declaração genérica para as funções genéricas que suportam múltiplos tipos é mostrada aqui:

```
template <class Ttipo1, class Ttipo2, .... class TtipoN>
   tipo-retorno nome-função ([lista parâmetros])
 {
   // corpo da função
 }
```

Ao declarar uma função genérica que suporta múltiplos tipos, você deverá evitar declarar tipos genéricos demais dentro da função, pois declarar tipos demais poderá criar mais confusão que soluções. Você também precisa ter a certeza de que sua função pode sempre derivar seu *tipo-retorno* a partir dos tipos genéricos que recebe como parâmetros.

1114 SOBRECARREGANDO EXPLICITAMENTE UMA FUNÇÃO GENÉRICA

Nas dicas anteriores, você aprendeu como definir funções genéricas. Também aprendeu que uma função genérica é essencialmente "de auto-sobrecarga" — em outras palavras, cria tantas versões para si mesma quantas são necessárias. No entanto, algumas vezes você irá querer sobrecarregar explicitamente um gabarito. Por exemplo, *você pode ter uma função soma generalizada, que atua diferentemente quando soma valores long.* Se você sobrecarregar uma função genérica, a função sobrecarregada ocultará a função genérica para as ocorrências dos valores especificados. O programa a seguir, *sobr_tmp.cpp,* sobrecarrega uma função de gabarito explicitamente:

```
#include <iostream.h>

template <class X> void permuta(X &a, X &b);
void permuta(int &a, int &b);

void main(void)
 {
   int i = 10, j = 20;
   float x = 10.1, y = 23.3;
   char a = 'x', b = 'z';

   cout << "Original i, j: " << i << " " << j << endl;
   cout << "Original x, y: " << x << " " << y << endl;
   cout << "Original a, b: " << a << " " << b << endl;
   permuta(i,j);  // sobrecarregou permuta explicitamente
   permuta(a,b);
   permuta(x,y);
   cout << "Permutei i, j: " << i << " " << j << endl;
   cout << "Permutei x, y: " << x << " " << y << endl;
   cout << "Permutei a, b: " << a << " " << b << endl;
 }

template <class X> void permuta(X &a, X &b)
 {
   X temp;

   temp = a;
   a = b;
```

```
    b = temp;
  }

void permuta(int &a, int &b)
  {
    int temp;

    temp = a;
    a = b;
    b = temp;
    cout << "Dentro da função permuta sobrecarregada." << endl;
  }
```

Quando você compilar e executar o programa *sobr_tmp.cpp*, sua tela exibirá a seguinte saída:

```
Original i, j: 10 20
Original x, y: 10.1 23.3
Original a, b: x z
Dentro da função permuta sobrecarregada
Permutou i, j: 20 10
Permutou x, y: 23.3 10.1
Permutou a, b: z x
C:\>
```

COMPREENDENDO AS RESTRIÇÕES NAS FUNÇÕES GENÉRICAS 1115

As funções genéricas são parecidas com as funções sobrecarregadas. No entanto, junto com o poder das funções genéricas, C++ também aplica mais restrições às funções genéricas do que às funções sobrecarregadas. Quando você sobrecarrega funções, suas funções sobrecarregadas podem efetuar ações diferentes dependendo do critério da sobrecarga. No entanto, quando você escrever uma função genérica, suas funções precisarão efetuar o mesmo processamento em todos os dados, independente do tipo. Por exemplo, no programa a seguir, *sem_temp.cpp*, você não pode substituir as funções sobrecarregadas com uma função genérica porque as atividades das funções sobrecarregadas diferem:

```
#include <iostream.h>
#include <math.h>

void nao_gabarito(int i)
  {
    cout << "O valor é: " << i << endl;
  }

void nao_gabarito(double d)
  {
    double parte_int;
    double parte_frac;

    parte_frac = modf(d, &parte_int);
    cout << "Parte fracionária: " << parte_frac << endl;
    cout << "Parte inteira: " << parte_int  << endl;
  }

void main(void)
  {
    nao_gabarito(1);
    nao_gabarito(12.2);
  }
```

1116 USANDO UMA FUNÇÃO GENÉRICA

Como você pode ver, as funções genéricas são um dos aspectos mais úteis de C++. Você pode aplicar função genérica a todos os tipos de situações. A qualquer tempo que você tiver uma função que defina um algoritmo generalizável, poderá tornar a função uma função de gabarito. Após você ter criado a função, poderá usar a função genérica com qualquer tipo de dado sem ter que reescrever a função. Em dicas posteriores você aprenderá como usar gabaritos para definir classes. Na dica a seguir você criará uma função genérica do algoritmo bolha. Por ora, certifique-se de compreender claramente o processamento da função genérica antes de avançar para dicas posteriores. Para compreender melhor o processamento da função genérica, considere o seguinte programa simples, *exib_txt.cpp*, que escreve uma função de saída genérica:

```
#include <iostream.h>

template <class T1, class T2> void exemplo(T1 x, T2 y);

void main(void)
 {
   exemplo(10, "Oi");
   exemplo(0.23, 10L);
   exemplo("C/C++","do Jamsa");
 }

template <class T1, class T2> void exemplo(T1 x, T2 y)
 {
   cout << x << " " << y << endl;
 }
```

Quando você compilar e executar o programa *exib_txt.cpp*, o compilador criará três versões da função *exemplo:* uma que aceita um inteiro e uma string, uma que aceita um *float* e um *long* e uma que aceita duas strings. No entanto, quando você executar o programa *exib_txt.cpp*, o fato de o compilador ter criado três versões da função (em vez de o próprio programa incluir três versões da função) é invisível para o usuário, que verá somente a seguinte saída:

```
10 oi
0.23 10
C/C++ do Jamsa
C:\>
```

No entanto, para efetuar esses mesmos passos sem uma função genérica, você teria que escrever um programa tal como *exib_nao.cpp*, mostrado aqui:

```
#include <iostream.h>

void exemplo(int x, char *y);
void exemplo(float x, long y);
void exemplo(char *x, char *y);

void main(void)
 {
   exemplo(10, "oi");
   exemplo(0.23, 10L);
   exemplo("C/C++ ", "do Jamsa");
 }

void exemplo(int x, char *y)
 {
   cout << x << " " << y << endl;
 }

void exemplo(float x, long y)
 {
```

```
  cout << x << " " << y << endl;
}

void exemplo(char *x, char *y)
{
  cout << x << " " << y << endl;
}
```

USANDO UMA FUNÇÃO GENÉRICA DO ALGORITMO DA BOLHA 1117

Como você aprendeu, as funções genéricas podem ser úteis para tornar seus programas mais fáceis de compreender e podem poupar um tempo considerável na programação. Para compreender melhor a utilidade da função genérica, considere um programa que você escreveu anteriormente e determine como uma função genérica poderia torná-lo mais útil. Por exemplo, nas Dicas 492 e 493, você criou uma função simples do algoritmo da bolha que trabalhava somente com valores inteiros. Essa função seria significativamente mais útil se usasse um gabarito genérico, em vez de uma função específica. Você escreveu a função original para aceitar uma matriz de inteiros para classificação, como a seguir:

```
void algoritmo_bolha(int matriz[], int tamanho)
{
  int temp, i, j;

  for (i = 0; i < tamanho; i++)
    for (j = 0; j < tamanho; j++)
      if (matriz[i] < matriz[j])
       {
         temp = matriz[i];
         matriz[i] = matriz[j];
         matriz[j] = temp;
       }
}
```

Usando seu conhecimento das funções genéricas, você pode agora escrever o código para o algoritmo da bolha para que ele trate todos os tipos de valores, não apenas matrizes *int*, como mostrado aqui dentro do programa *bolha.cpp:*

```
#include <iostream.h>

template <class X> void algor_bolha(X *items, int tamanho)
template <class X> void exibe_itens(X *items, int tamanho)

void main(void)
{
  int i_matriz[7] =    { 7, 5, 4, 3, 9, 8, 6 };
  double d_matriz[5] = { 4.2, 2.5, -0.9, 100.2, 3.0 };

  cout << "Matriz de ints não-classificada: " << endl;
  exibe_itens(i_matriz, 7);
  cout << "Matriz de doubles não-classificada: " << endl;
  exibe_itens(d_matriz, 5);
  algor_bolha(i_matriz, 7);
  algor_bolha(d_matriz, 5);
  cout << "Matriz de ints classificada: " << endl;
  exibe_itens(i_matriz, 7);
  cout << "Matriz de doubles classificada: " << endl;
  exibe_itens(d_matriz, 5);
}

template <class X> void algor_bolha(X *items, int tamanho)
```

```
{
  register int i, j;
  X temp;

  for (i = 1; i < tamanho; i++)
   for (j = tamanho-1; j >= i; j--)
     if (items[j-1] > items[j])
       {
         temp = items[j-1];
         items[j-1] = items[j];
         items[j] = temp;
       }
 }

template <class X> void exibe_items(X *items, int tamanho)
 {
   int i;

   for(i=0; i < tamanho; i++)
      cout << items[i] << ", ";
   cout << endl;
 }
```

Quando você compilar e executar o programa *bolha.cpp*, sua tela exibirá a seguinte saída:

```
Matriz de ints não-classificada:
7, 5, 4, 3, 9, 8, 6,
Matriz de doubles não-classificada:
4.2, 2.5, -0.9, 100.2, 3,
Matriz de ints classificada:
3, 4, 5, 6, 7, 8, 9
Matriz de doubles classificada:
-0.9, 2.5, 3, 4.2, 100.2,
C:\>
```

1118 USANDO FUNÇÕES GENÉRICAS PARA COMPACTAR UMA MATRIZ

À medida que seus programas forem trabalhando com matrizes, algumas vezes você desejará remover elementos do meio de uma matriz e mover os elementos restantes na matriz "para a frente" para preencher o espaço recém-criado — um processo chamado *compactação*. Um modo de evitar a necessidade de compactação de matriz é usar uma lista ligada, sobre a qual você aprendeu anteriormente. No entanto, você poderá achar mais conveniente trabalhar com uma matriz. O programa a seguir, *compacta.cpp*, usa uma função genérica para compactar matrizes de múltiplos tipos:

```
#include <iostream.h>

template <class X> void compacta(X *items, int conta, int inicio, int fim)
template <class X> void exibe_itens(X *items, int tamanho)

void main(void)
 {
   int nums[7]  = {0, 1, 2, 3, 4, 5, 6};
   char str[18] = "Funções Genéricas";

   cout << "Matriz de ints não-compactada: ";
   exibe_itens(nums, 7);
   cout << "Matriz de chars não-compactada: ";
   exibe_itens(str, 18);
   compacta(nums, 7, 2, 4);
```

```
    compacta(str, 18, 6, 10);
    cout << "Matriz de ints compactada: ";
    exibe_itens(nums, 7);
    cout << "Matriz de chars compactada: ";
    exibe_itens(str, 18);
  }

template <class X> void compacta(X *items, int conta, int inicio, int fim)
  {
    register int i;
    for(i = fim+i; i < conta; i++, inicio++)
       items[inicio] = items[i];
    for( ; inicio<conta; inicio++)
       items[inicio] = (X) 0;
  }

template <class X> void exibe_itens(X *items, int tamanho)
  {
    int i;
    for(i = 0; i < tamanho; i++)
       cout << items[i];
    cout << fim1;
  }
```

No exemplo anterior, a função *compacta* preenche os elementos restantes dentro da matriz com zero. Quando você compilar e executar o programa *compacta.cpp*, sua tela exibirá a seguinte saída:

```
Matriz de ints não-compactada: 0123456
Matriz de chars não-compactada: Funções Genéricas
Matriz de ints compactada: 0100000
Matriz de ints compactada: Funçõesséricas
C:\>
```

ONDE COLOCAR OS GABARITOS 1119

Os gabaritos de C++ lhe permitem reduzir sua programação para as funções que diferem somente nos tipos de parâmetros e de retorno. À medida que você cria gabaritos, deve colocá-los em arquivos de cabeçalho de nomes representativos para poder reutilizá-los em outros programas. Como é bem provável que você crie apenas alguns gabaritos inicialmente, você pode querer usar o arquivo de cabeçalho *gabarito.h*. À medida que criar mais gabaritos, poderá atribuí-los a outros arquivos de cabeçalho, com base em suas funções.

OS GABARITOS TAMBÉM ELIMINAM AS CLASSES DUPLICADAS 1120

Os gabaritos de C++ podem reduzir sua programação para as funções que diferem somente nos tipos de parâmetros e valor de retorno. Seus programas também podem usar gabaritos para eliminar as classes similares. Por exemplo, considere as seguintes classes:

```
class CurtaDistancia  {
  public:
    CurtaDistancia (int distancia) { CurtaDistancia::distancia = distancia; };
    ExibeDistancia(void)
        { cout << "A distância é " << distancia << " Km" << endl; };
  private:
    int distancia;
};

class LongaDistancia  {
  public:
    LongaDistancia(long distancia) { LongaDistancia::distancia = distancia; };
```

```
    ExibeDistancia(void)
        { cout << "A distância é " << distancia << " Km" << endl; };
  private:
     long distancia;
};
```

Ambas as classes efetuam processamento similar; a única diferença é que a classe *LongaDistancia* mantém os valores do tipo *long int*, e a classe *CurtaDistancia* somente mantém valores do tipo *int*. Como as classes são fundamentalmente idênticas e efetuam o mesmo processamento, mas diferem apenas nos valores que mantêm, elas são boas candidatas a uma única classe genérica.

O programa a seguir, *classtmp.cpp*, combina os dois tipos de classes, *LongaDistancia* e *CurtaDistancia*, em uma classe *Distancia* genérica:

```
#include <iostream.h>

template<class T> class Distancia  {
  public:
    Distancia(T distancia);
    void exibe_distancia(void)
      {cout << "A distância é " << distancia << " Km\n"; };
  private:
    T distancia;
};

template<class T>  Distancia<T>::Distancia(T distancia)
{ Distancia::distancia = distancia; };

void main(void)
 {
    Distancia<int> curta_distancia(100);
    Distancia<long> longa_distancia(2000000L);

    curta_distancia.exibe_distancia();
    longa_distancia.exibe_distancia();
 }
```

Quando você compilar e executar o programa *classtmp.cpp*, o compilador C++ criará classes usando os tipos corretos. Como você verá na dica a seguir, os gabaritos de classe, tal como *Distancia*, são freqüentemente chamados de *classe genérica* ou *gerador de classe*. Quando você compilar e executar o programa *classtmp.cpp*, sua tela exibirá a seguinte saída:

```
A distância é 100 Km
A distância é 2000000 Km
C:\>
```

1121 COMPREENDENDO AS CLASSES GENÉRICAS

Como viu na dica anterior, C++ lhe permite usar gabaritos para definir as classes genéricas. Quando você especifica um gabarito de classe, sempre precisa especificar o nome correspondente, seguido pelos sinais de menor e maior e um tipo (ou o marcador de lugar do tipo ou o tipo real), como mostrado aqui:

```
Distancia<T>
Distancia<int>
```

Examinando os programas C++ que usam gabaritos de classe, você poderá encontrar programas que definem gabaritos que aceitam parâmetros, como a seguir:

```
gabarito<class T, int tam_matriz = 64> class AlgumaClasse {
   // Comandos
};
```

Neste caso, o gabarito não somente especifica um marcador de lugar do tipo como também especifica um parâmetro que o programa pode usar dentro do gabarito. Quando o programa mais tarde usar o gabarito, ele poderá passar um valor de parâmetro para o gabarito, como segue:

```
AlgumaClasse<int, 1024> esta_ocorrencia;
```

USANDO CLASSES GENÉRICAS 1122

Em dicas anteriores você criou e usou várias funções genéricas. No entanto, como aprendeu na dica anterior, C++ também suporta classes genéricas. Criar uma classe genérica, embora mais difícil que criar uma classe tipada, também aumenta seu poder de programação. Você pode usar classes genéricas para permitir que seus programas acessem um conjunto muito maior de tipos de valor sem muita codificação adicional. Como você aprenderá em dicas posteriores, a Biblioteca de Gabaritos Padrões (STL) é construída com base nas classes genéricas. Para compreender melhor as classes genéricas, considere o programa a seguir, *gen_pilh.cpp:*

```
#include <iostream.h>

const int TAMANHO = 100;

template <class PTipo> class pilha  {
   PTipo plh[TAMANHO];
   int topo;
 public:
   pilha(void);
   ~pilha(void);
   void push(PTipo i);
   PTipo pop(void);
 };

template <class PTipo> pilha<PTipo>::pilha(void)
 {
   topo = 0;
   cout << "Pilha inicializada." << endl;
 }

template <class PTipo> pilha<PTipo>::~pilha()
 {
   cout << "Pilha destruída." << endl;
 }

template <class PTipo> void pilha<PTipo>::push(PTipo i)
 {
   if(topo == TAMANHO)
    {
      cout << "A pilha está cheia." << endl;
      return;
    }
   plh[topo++] = i;
 }

template <class PTipo> PTipo pilha<PTipo>::pop(void)
 {
   if(topo == 0)
    {
      cout << "Erro na pilha." << endl;
      return 0;
    }
   return plh[--topo];
 }

void main(void)
```

```
{
  pilha<int> a;
  pilha<double> b;
  pilha<char> c;
  int i;

  a.push(1);
  a.push(2);
  b.push(99.3);
  b.push(-12.23);

  cout << a.pop() << " ";
  cout << a.pop() << " ";
  cout << b.pop() << " ";
  cout << b.pop() << endl;

  for(i = 0; i < 10; i++)
    c.push((char) 'A' + i);

  for(i = 0; i < 10; i++)
    cout << c.pop();
  cout << endl;
}
```

O programa *gen_pilh.cpp* define a classe genérica *pilha*, que mantém uma matriz de 100 elementos de seu tipo definido e um valor inteiro que você pode usar para acessar os elementos dentro da pilha. Como você viu dentro de suas definições anteriores da classe *pilha*, o programa também define as funções-membro *push* e *pop* que você pode usar para colocar e retirar dados de um objeto *pilha*. Quando o programa inicia sua execução, ele cria três ocorrências da classe genérica *pilha:* uma pilha *int*, uma pilha *double* e uma pilha *char*. O restante do código manipula cada pilha, uma de cada vez. Quando você compilar e executar o programa *gen_pilh.cpp*, sua tela exibirá a seguinte saída:

```
Pilha inicializada.
Pilha inicializada.
Pilha inicializada.
2 1 -12.33 99.3
[illegible]
Pilha destruída.
Pilha destruída.
Pilha destruída.
C:\>
```

1123 CRIANDO UMA CLASSE GENÉRICA COM DOIS TIPOS DE DADOS GENÉRICOS

Seus programas podem declarar funções genéricas que suportam múltiplos tipos de dados genéricos. Similarmente, seus programas podem declarar classes genéricas que suportam múltiplos tipos de dados genéricos. Para declarar uma classe genérica que suporta múltiplos tipos de dados genéricos, você usará uma forma expandida da declaração de classe genérica padrão, como mostrado aqui:

```
template <class T1, class T2, ... class TN> class nome_classe  {
        // definições de membro
}
```

Os marcadores de lugar *T1*, *T2* e *TN* representam os tipos genéricos que sua classe aceitará. Uma classe pode, em teoria, aceitar um número infinito de tipos genéricos dentro de sua definição. No entanto, exatamente como com as funções genéricas que suportam múltiplos tipos, você deve ser cuidadoso para não definir tipos genéricos demais em suas classes, pois fazer isso criará confusão dentro de seus programas. Para compreender melhor como você declarará as classes genéricas que suportam dois ou mais tipos de dados genéricos, considere o programa *duas_gen.cpp*, como segue:

```
#include <iostream.h>

template <class T1, class T2> class dois_gen {
   T1 i;
   T2 j;
 public:
   dois_gen(T1 a, T2 b)
      { i=a; j=b;}
   void exibe(void)
      { cout << i << " " << j << endl; }
 };

void main(void)
 {
   dois_gen<int, double> obj1(10, 0.23);
   dois_gen<char, char *> obj2('X', "Isto é um teste.");

   obj1.exibe();
   obj2.exibe();
 }
```

CRIANDO UM MANIPULADOR PARAMETRIZADO 1124

Você aprendeu na Dica 1001 que usará classes genéricas para criar manipuladores parametrizados. Quando você cria um manipulador parametrizado, precisa incluir o arquivo *iomanip.h* dentro de seu programa. O arquivo *iomanip.h* define três classes genéricas: *omanip*, *imanip* e *smanip*. Em geral, quando você precisar criar um manipulador que receberá um argumento, seu programa precisará criar duas funções de manipuladores sobrecarregadas. A primeira precisa definir dois parâmetros: uma referência para o canal e um parâmetro que a primeira função passa para a segunda função (a função genérica). A função genérica aceita um único parâmetro e gera uma chamada para a primeira função. Para compreender melhor o relacionamento entre as duas funções, considere a forma generalizada do manipulador de saída parametrizado, como mostrado aqui:

```
ostream &nome-manip(ostream &canal, tipo param)
 {
   // seu código aqui
   return canal;
 }

// Função genérica sobrecarregada
omanip<tipo> nome-manip(tipo param)
 {
   return omanip<tipo> (nome-manip, param);
 }
```

Na forma generalizada, *nome-manip* é o nome do manipulador, e *tipo* especifica o tipo do parâmetro que o manipulador usa. Como *omanip* é também uma classe genérica, *tipo* também é o tipo de dado sobre o qual o objeto *omanip* (que a segunda função sobrecarregada retorna) atua.

Claramente, escrever manipuladores parametrizados é uma tarefa difícil e normalmente confusa. Para compreender melhor como você usará os manipuladores parametrizados dentro de seus programas, considere o programa *man_para.cpp*:

```
#include <iostream.h>
#include <iomanip.h>

ostream &recua(ostream &canal, int tamanho)
 {
   register int i;

   for(i = 0; i < tamanho; i++)
      cout << " ";
```

```
  return canal;
 }

omanip<int> recua(int tamanho)
 {
  return omanip<int>(recua, tamanho);
 }

void main(void)
 {
  cout << recua(10) << "Isto é um teste." << endl;
  cout << recua(20) << "do novo manipulador recua." << endl;
  cout << recua(5) << "Ele funciona!" << endl;
 }
```

Quando você compilar e executar o programa *man_para.cpp*, ele exibirá a saída mostrada aqui:

```
          Isto é um teste.
                    do novo manipulador recua.
    Ele funciona!
C:\>
```

1125 CRIANDO UMA CLASSE DE MATRIZ GENÉRICA

Nas dicas anteriores vimos como criar classes e funções genéricas e atividades que os gabaritos permitem ou o ajudam a executar. Como aprendeu em muitas dicas anteriores neste livro, geralmente você criará uma matriz de tipos de classes dentro da maioria dos programas. Portanto, você precisa compreender como manipular uma classe genérica para inicializar uma matriz dessa classe.

Criar uma matriz genérica é tão fácil quanto criar uma matriz de um tipo normal. Por exemplo, se todas as suas matrizes são do mesmo tamanho, você poderia usar uma classe genérica sem um parâmetro para declarar uma matriz de inteiros, como mostrado aqui:

```
umtipo<int> int_matriz;
```

Usar uma classe genérica para criar matrizes lhe permite operar em matrizes diferentemente. Como você já aprendeu, seus programas podem sobrecarregar o operador [] para criar "matrizes seguras" — matrizes cujos limites seus programas não podem ultrapassar, entre outros critérios para as matrizes seguras. Se você criar uma classe genérica e sobrecarregar o operador [] relativo a essa classe, poderá forçar todas as matrizes dentro de seus programas para as matrizes seguras. O programa a seguir, *gen_segu.cpp*, usa uma classe genérica e um operador [] sobrecarregado para criar matrizes seguras:

```
#include <iostream.h>
#include "stdlib.h"

const int TAMANHO = 10;

template <class UmTipo> class umtipo {
   UmTipo a[TAMANHO];
 public:
   umtipo(void)
    {
      int i;

      for(i = 0; i < TAMANHO; i++)
         a[i] = i;
    }
   UmTipo &operator[](int i);
 };

template <class UmTipo> UmTipo &umtipo<UmTipo>::operator[](int i)
 {
```

```
    if(i < 0 || i > TAMANHO-1)
     {
       cout << endl << "O valor de índice ";
       cout << i << " está fora dos limites." << endl;
     }
    return a[i];
  }

void main(void)
 {
   umtipo<int> int_matriz;
   umtipo<double> double_matriz;
   int i;

   cout << "Matriz de inteiros: ";
   for(i = 0; i < TAMANHO; i++)
      int_matriz[i] = i;

   for(i = 0; i < TAMANHO; i++)
      cout << int_matriz[i] << " ";
   cout << endl;

   cout << "Matriz de doubles: ";
   cout.precision(2);
   for(i = 0; i < TAMANHO; i++)
      double_matriz[i] = (double)i/3;

   for(i = 0; i < TAMANHO; i++)
      cout << double_matriz[i] << " ";
   cout << endl;

   int_matriz[12] = 100; // Chama o operador de matriz sobrecarregado
 }
```

O programa *gen_segu.cpp* cria a classe genérica *umtipo* e sobrecarrega o operador [] dentro da classe. A função sobrecarregada verifica para garantir que o usuário não tente acessar ou inicializar qualquer elemento da matriz que esteja fora dos limites anteriormente definidos da matriz. Quando o programa *gen_segu cpp* é executado, ele cria duas matrizes (uma *int* e outra *double*), preenche as matrizes e exibe seus valores. Finalmente, o último comando tenta inicializar um valor fora dos limites da matriz. Quando você compilar e executar o programa *gen_segu.cpp*, sua tela exibirá a seguinte saída:

```
Matriz de inteiros: 0 1 2 3 4 5 6 7 8 9
Matriz de doubles: 0 0.33 0.67 1 1.3 1.7 2 2.3 2.7 3
O valor de índice 12 está fora dos limites.
C:\>
```

COMPREENDENDO O TRATAMENTO DAS EXCEÇÕES 1126

À medida que você for criando bibliotecas, algumas vezes poderá prever de antemão os tipos de erros de execução que um programa encontrará ao trabalhar com a classe (tal como sobrescrever os limites da matriz ou passar um valor grande demais). Infelizmente, algumas vezes você não pode escrever o código para interceptar esses erros quando eles ocorrem dentro de outro programa. Para esses casos, muitos compiladores C++ suportam as *rotinas de tratamento de exceção*. Em geral, uma rotina de tratamento de exceção é o software que executa quando tal erro ocorre. Dentro do código de biblioteca da classe, você testaria o erro possível. Se o erro ocorreu, você geraria uma exceção. O programa do usuário que experimentou o erro é responsável por *pegar* e tratar a exceção — o que significa que o programa precisa fornecer software de tratamento da exceção. Para suportar o tratamento da exceção, a linguagem C++ oferece as palavras-chave mostradas na Tabela 1126.

Tabela 1126 Palavras-chave de tratamento de exceção de C++.

Palavra-Chave	Significado
catch	Pega a exceção gerada
throw	Inicia uma rotina de tratamento da exceção
try	Tenta uma operação para testar uma possível exceção

1127 COMPREENDENDO A FORMA DE TRATAMENTO BÁSICO DA EXCEÇÃO

Como você aprendeu na dica anterior, C++ lhe oferece vários comandos que seus programas podem usar juntos para pegar as exceções dentro do código do seu programa. Os três componentes básicos de toda rotina de tratamento de exceção são os comandos *try*, *catch* e *throw*. Quando seus programas efetuam o processamento da exceção, você precisa incluir os comandos que quer monitorar para as exceções (isto é, erros) dentro do bloco *try*. Se um comando processa incorretamente, você precisa gerar (*throw*, lançar) um erro apropriado para a ação da função. O programa do usuário pega o comando *throw* dentro do bloco *catch*, que efetua o tratamento da exceção real. A forma generalizada do bloco de tratamento da exceção é mostrada aqui:

```
try  {
try block
   // if(erro)  throw valor-exceção;
 }
catch(tipo-exceção nome-variável)   {
   // processamento do tratamento da exceção
 }
```

Dentro da forma generalizada da rotina de tratamento de exceção, o *valor da exceção lançada* precisa ser igual ao *tipo da exceção apanhada*, como você aprenderá em dicas posteriores. Na dica a seguir, você escreverá uma rotina de tratamento de exceção simples com base na forma generalizada mostrada nesta dica.

1128 ESCREVENDO UMA ROTINA DE TRATAMENTO DE EXCEÇÃO SIMPLES

Cada rotina de tratamento de exceção que escreve dentro de seus programas incluirá um comando *try*, um ou mais comandos *throw*, e um ou mais comandos *catch*, como já foi visto. Para compreender melhor o processamento que uma rotina de tratamento de exceção efetua, considere o programa a seguir, *simples.cpp*, que usa uma rotina de tratamento de exceção para lançar e processar um erro dentro de *main*:

```
#include <iostream.h>

void main(void)
 {
   cout << "Início" << endl;
   try {
      cout << "Dentro do bloco de teste." << endl;
      throw 100;
      cout << "Isto não será executado.";
    }
   catch(int i) {
      cout << "Peguei uma exceção -- o valor é: ";
      cout << i << endl;
    }
   cout << "Fim";
 }
```

O programa *simples.cpp* implementa um único bloco *try-catch*. Em vez de esperar que o programa cometa um erro, o programa usa o comando *throw* (sobre o qual você aprenderá na dica a seguir) para causar um erro

Após o bloco *try* gerar o erro, o bloco *catch* o pega e processa o valor que o comando *throw* passa. Quando você compilar e executar o programa *simples.cpp*, sua tela exibirá a seguinte saída:

```
Início
Dentro do bloco de teste
Peguei uma exceção -- o valor é 100
Fim
C:\>
```

COMPREENDENDO O COMANDO THROW 1129

Você aprendeu que o terceiro componente de uma rotina de tratamento de exceção é o comando *throw*, o que chama o comando *catch*. Você precisa incluir pelo menos um comando *throw* dentro de um bloco *try* para a rotina de tratamento de exceção efetuar qualquer processamento de valor. Como você viu em dicas anteriores, o formato generalizado do comando *throw* é mostrado aqui:

```
throw exceção;
```

A *exceção* precisa corresponder ao tipo de *exceção* que o bloco *catch* especifica. À medida que seus programas se tornarem mais complexos, e especialmente à medida que você criar mais programas no Windows, verá que pode usar blocos *try-catch* (normalmente chamados de blocos de *teste*) com muitas funções para criar programas mais estáveis e seguros.

AS EXCEÇÕES SÃO DE UM TIPO ESPECÍFICO 1130

Você viu que seus programas podem efetuar atividades de tratamento de exceção usando um *bloco try*. O primeiro bloco *try* que você viu (na Dica 1127) pegou uma exceção inteira. No entanto, seus comandos *catch* podem pegar qualquer tipo de exceção. Como resultado, você precisa ser cuidadoso para que a exceção que seu programa gera dentro do bloco *try* seja igual ao tipo que o bloco *catch* especifica. Por exemplo, o programa a seguir, *catch_d.cpp*, pega uma exceção *double*, enquanto o bloco *try* lança uma exceção inteira:

```
#include <iostream.h>

void main(void)
 {
   cout << "Início" << endl;
   try {
      cout << "Dentro do bloco de teste." << endl;
      throw 100;
      cout << "Isso não será executado.";
    }
   catch(double d) {
      cout << "Peguei uma exceção double -- o valor é: ";
      cout << d << endl;
    }
   cout << "Fim";
 }
```

Dependendo do seu compilador, quando você compilar e executar o programa *catch_d.cpp*, poderá receber uma advertência do compilador dizendo que o comando é um código não-alcançável. Se receber essa mensagem durante a compilação, verifique seu código atentamente para localizar erros lógicos, tais como aquele no programa *catch_d.cpp*. Se você receber essa mensagem de erro e mesmo assim rodar o programa, ele gerará saída similar ao programa *catch_d.cpp*, como a seguir:

```
Início
Dentro do bloco de teste.
Abnormal program termination.
C:\>
```

1131 LANÇANDO EXCEÇÕES A PARTIR DE UMA FUNÇÃO DENTRO DE UM BLOCO TRY

Como você aprendeu, seus programas podem efetuar atividades de tratamento de exceção usando um bloco *try*. No entanto, seus programas podem frequentemente chamar funções a partir de dentro de um bloco *try*. Quando seus programas chamam funções a partir de dentro de um bloco *try*, C++ passará a exceção para o bloco *try* fora da função (desde que não haja um segundo bloco *try* dentro da função, como você aprenderá em dicas posteriores). O programa a seguir, *sai_func.cpp*, usa um bloco *try* dentro de *main* para chamar a função *TrataExcecao*

```
#include <iostream.h>

void TrataExcecao(int teste)
 {
    cout << "Dentro de TrataExcecao, teste é:" << teste << endl;
    if(teste) throw
    teste;
 }

void main(void)
 {
   cout << "Início: " << endl;
   try {
      cout << "Dentro do bloco try." << endl;
      TrataExcecao(1);
      TrataExcecao(2);
      TrataExcecao(0);
    }
   catch(int i) {
      cout << "Peguei uma exceção. O valor é: ";
      cout << i << endl;
    }
   cout << "Fim";
 }
```

Quando você compilar e executar o programa *sai_func.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Dentro do bloco try.
Dentro de TrataExcecao, teste é: 1
Peguei uma exceção. O valor é: 1
Fim
C:\>
```

1132 LOCALIZANDO UM BLOCO TRY PARA UMA FUNÇÃO

Seus programas podem usar blocos *try* para pegar exceções durante o processamento. Em dicas anteriores, você criou um bloco *try* simples dentro de *main* e criou um bloco *try* dentro de *main* que chamou uma função a partir de dentro de si mesma. No entanto, você também pode usar blocos *try* localmente dentro de funções. Quando você localiza um bloco *try* para uma função, C++ reinicializa o bloco toda vez que você entra na função. O programa a seguir, *fun_cat.cpp*, exibe como você pode localizar blocos *try* dentro de funções:

```
#include <iostream.h>

void TrataExcecao(int teste)
 {
   try {
      if(teste) throw teste;
    }
```

```
    catch(int i)
     {
       cout << "Peguei a exceção #: " << i << endl;
     }
  }

void main(void)
  {
    cout << "Início: " << endl;
    TrataExcecao(1);
    TrataExcecao(2);
    TrataExcecao(0);
    TrataExcecao(3);
    cout << "Fim";
  }
```

Quando você compilar e executar o programa *fun_cat.cpp*, sua tela exibirá a seguinte saída (observe que a função somente lança três exceções porque a terceira chamada, com seu valor zero, é avaliada como falsa):

```
Início:
Peguei a exceção #: 1
Peguei a exceção #: 2
Peguei a exceção #: 3
Fim
C:\>
```

COMPREENDENDO QUANDO O PROGRAMA EXECUTA CATCH 1133

Vimos que seus programas podem usar uma sequência *try-catch* para controlar o tratamento da exceção e se proteger contra a finalização anormal do programa. Em dicas anteriores, você aprendeu como pegar as exceções dentro de um bloco *catch*. No entanto, você pode estar preocupado com seus programas que executarão comandos dentro de um bloco *catch* mesmo se o programa não gerar uma exceção. Porém, seus programas somente executarão os comandos dentro de blocos *catch* se o programa gerar uma exceção dentro do bloco *try* imediatamente precedente. O programa a seguir, *semcatch.cpp*, ilustra como seus programas pularão os comandos dentro de um bloco *catch* a não ser que o programa gere uma exceção:

```
#include <iostream.h>

void main(void)
  {
    cout << "Início" << endl;
    try
     {
       cout << "Dentro do bloco try." << endl;
       cout << "Ainda dentro do bloco try." << endl;
     }
    catch(int i)
     {
       cout << "Peguei uma exceção -- o valor é: " << endl;
       cout << i << endl;
     }
    cout << "Fim";
  }
```

Quando você compilar e executar o programa semcatch.cpp, ele pulará os comandos que o bloco catch contém, pois o programa não lançou uma exceção no bloco *try*. O programa *semcatch.cpp* exibirá a seguinte saída na tela:

```
Início
Dentro do bloco try.
Ainda dentro do bloco try.
Fim
C:\>
```

1134 USANDO MÚLTIPLOS COMANDOS CATCH COM UM ÚNICO BLOCO TRY

À medida que suas exceções se tornarem mais complexas, algumas vezes um único bloco *try* lançará exceções de múltiplos tipos. Dentro de seus programas, você irá querer construir sua rotina de tratamento de exceção para que ela permita a captura de múltiplas exceções. Quando você precisar pegar múltiplas exceções, usará o seguinte formato generalizado:

```
try
   {
     // comandos
   }
catch(tipo1)
   {
   // processamento da exceção
   }
catch(tipo2)
   {
   // processamento da exceção
   }
.
.
.
catch(tipoN)
   {
   // processamento da exceção
   }
```

Como você aprenderá em dicas posteriores, seus comandos *catch* podem reconhecer qualquer tipo retornado, não apenas os tipos básicos que C++ suporta. Na verdade, seus comandos *catch* podem até pegar tipos definidos lançados pelo usuário. O programa a seguir, *catch_3.cpp*, usa múltiplos comandos *catch* para pegar várias exceções de diferentes tipos:

```
#include <iostream.h>

void TrataExcecao(int teste)
 {
   try
   {
      if(teste==0)
      throw teste;
      if(teste==1)
      throw "String";
      if(teste==2)
      throw 123.23;
    }
   catch(int i)
    {
      cout << "Peguei a exceção #: " << i << endl;
    }
   catch(char *str)
    {
      cout << "Peguei exceção de string: " << str << endl;
    }
   catch(double d)
    {
      cout << "Peguei exceção #: " << d << endl;
    }
 }
```

```
void main(void)
 {
   cout << "Início: " << endl;
   TrataExcecao(0);
   TrataExcecao(1);
   TrataExcecao(2);
   cout << "Fim";
 }
```

O programa *catch_3.cpp* usa uma série de comandos *if* dentro do bloco *try* para lançar três exceções diferentes em três chamadas de função diferentes. Quando você compilar e executar o programa *catch_3.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Peguei a exceção #: 0
Peguei exceção de string: String
Peguei exceção #: 123.23
Fim
C:\>
```

USANDO O OPERADOR RETICÊNCIAS (...) COM AS EXCEÇÕES 1135

Como você já viu em dicas anteriores, seus programas podem pegar exceções dentro de múltiplos blocos *try* ou usar múltiplos comandos *catch* dentro de um único bloco *try*. No entanto, seus programas também podem usar o operador *reticências* para pegar genericamente todos os erros que ocorrem dentro de um único bloco *try*. Para pegar todos os erros dentro de um único bloco *try*, construa o bloco *try* no seguinte formato generalizado:

```
try
 {
   // comandos
 }
catch(...)
 {
   // processamento da exceção
 }
```

PEGANDO TODAS AS EXCEÇÕES DENTRO DE UM ÚNICO BLOCO TRY 1136

Como você aprendeu na dica anterior, seus programas podem usar o operador *reticências* para pegar múltiplas exceções. Como você aprenderá na dica a seguir, poderá usar o operador *reticências* junto com o bloco *try-catch* padrão. No entanto, você também usará o operador *reticências* sozinho para pegar múltiplas exceções de tipos desconhecidos ou diferentes. O programa a seguir, *catch_ml.cpp*, usa o operador *reticências* para pegar três exceções de três tipos diferentes:

```
#include <iostream.h>

void TrataExcecao(int teste)
 {
   try
   {
      if(teste==0)
      throw teste;
      if(teste==1)
      throw 'a';
      if(teste==2)
      throw 123.23;
```

```
    }
   catch(...)
    {
      cout << "Peguei uma." << endl;
    }
 }

void main(void)
 {
   cout << "Início: " << endl;
   TrataExcecao(0);
   TrataExcecao(1);
   TrataExcecao(2);
   cout << "Fim";
 }
```

Observe que, ao contrário do programa *catch_3.cpp* da Dica 1134, o programa *catch_ml.cpp* usa somente um único comando *catch* para pegar todos os três erros. Quando você compilar e executar o programa *catch_ml.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Peguei uma.
Peguei uma.
Peguei uma.
Fim
C:\>
```

Observe que, no programa *catch_ml.cpp*, seu bloco *catch* que usa o operador *reticências* não pode discernir o tipo de exceção que o programa lança — o que significa que seu processamento dentro do bloco *catch* precisa ser independente de erro. A dica a seguir explicará como incluir o tratamento de erro dependente de erro e independente de erro dentro de um único bloco.

1137 PEGANDO EXCEÇÕES EXPLÍCITAS E GENÉRICAS EM UM ÚNICO BLOCO TRY

Você aprendeu que pode usar o operador *reticências* para pegar múltiplas exceções de tipos desconhecidos. No entanto, mais freqüentemente, desejará que seus programas peguem exceções explícitas e respondam às exceções de modos específicos. Se você estiver processando exceções específicas e explícitas dentro de um determinado bloco *try*, poderá adicionalmente querer usar o operador *reticências* para pegar todas as exceções que não são do tipo ou tipos esperados. Como C++ lhe permite pegar múltiplos tipos de exceção dentro de um determinado bloco *try*, você pode criar facilmente seqüências de exceção que lhe permitam pegar exceções explícitas e genéticas e tratá-las de forma diferente. O programa a seguir, *exp_ines.cpp*, pega tanto exceções inteiras e desconhecidas dentro de um único bloco *try*:

```
#include <iostream.h>

void TrataExcecao(int teste)
 {
   try
    {
      if(teste==0)
      throw teste;
      if(teste==1)
      throw 'a';
      if(teste==2)
      throw 123.23;
    }
   catch(int i)
    {
```

```
      cout << "Peguei um inteiro." << endl;
    }
   catch(...)
    {
      cout << "Peguei uma." << endl;
    }
 }

void main(void)
 {
   cout << "Início: " << endl;
   TrataExcecao(0);
   TrataExcecao(1);
   TrataExcecao(2);
   cout << "Fim";
 }
```

O programa *exp_ines.cpp* pegará o *int* lançado dentro do bloco explícito, e o *double* e o *char* lançados dentro do bloco genérico. Quando você compilar e executar o programa *exp_ines.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Peguei um inteiro.
Peguei uma.
Peguei uma.
Fim
C:\>
```

RESTRINGINDO AS EXCEÇÕES 1138

À medida que seus programas se tornarem mais complexos, freqüentemente chamarão funções a partir de dentro do bloco *try*. Quando seus programas chamam funções a partir de dentro do bloco *try*, você pode restringir que tipo de exceções a função chamada pode lançar. Você também pode evitar que essa função lance qualquer exceção. Para restringir as exceções que suas funções podem lançar, você precisa acrescentar uma cláusula *throw* na definição da função. A forma geral da função de lançamento restrito é:

```
tipo-retorno nome-função(lista-argumento) throw(lista-tipo)
 {
   // Código da função aqui
 }
```

Quando você declarar uma função com a cláusula *throw*, ela somente poderá lançar aqueles tipos que você detalhar dentro da *lista-tipo*. Se a função lançar qualquer outro tipo de exceção, o programa terminará anormalmente (como você viu na Dica 1130 com o programa *catch_d.cpp*). Se você não quiser que uma função lance quaisquer exceções, use uma *lista-tipo* vazia. Para compreender melhor como os programas podem limitar as exceções lançadas de uma função, considere o seguinte programa, *fun_lanc.cpp*, que limita a função *TrataExcecao* a lançamentos do tipo *int*, *char* e *double*:

```
#include <iostream.h>

void TrataExcecao(int teste) throw(int, char, double)
 {
   if(teste==0)
   throw teste;
   if(teste==1)
   throw 'a';
   if(teste==2)
   throw 123.23;
 }

void main(void)
```

```
{
  cout << "Início: " << endl;
  try {
     TrataExcecao(0); // tenta passar 1 e 2 para diferentes respostas
   }
  catch(int i) {
     cout << "Peguei um inteiro." << endl;
   }
  catch(char c) {
     cout << "Peguei um caractere." << endl;
   }
  catch(double d) {
     cout << "Peguei um double." << endl;
   }
  cout << "Fim";
}
```

É importante que você compreenda que usar a cláusula *throw* na declaração de uma função somente limita as exceções que a função pode lançar de volta para o bloco *try* a partir de dentro do qual o programa chamou a função. Dentro da função, você ainda pode usar um bloco *try* para pegar qualquer exceção, mesmo as exceções não-listadas dentro da *lista-tipo*. Quando você compilar e executar o programa *fun_lanc.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Peguei um inteiro.
Fim
C:\>
```

A seguinte declaração da função *TrataExcecao* impedirá a função de lançar quaisquer exceções para o bloco chamador:

```
void TrataExcecao(int teste)  throw()
 {
   if(teste==0)
   throw teste;
   if(teste==1)
   throw 'a';
   if(teste==2)
   throw 123.23;
 }
```

Como o comando *throw* na declaração da função não inclui valores dentro da *lista-tipo*, a função não lançará exceções. Se você compilar e executar o arquivo *fun_nlan.cpp*, ele exibirá a seguinte saída na tela:

```
Início:
Abnormal program termination
C:\>
```

1139 RELANÇANDO UMA EXCEÇÃO

À medida que suas rotinas de tratamento de exceção vão se tornando mais complexas, algumas vezes seus programas precisarão relançar uma exceção a partir de dentro de uma rotina de tratamento de exceção. Se você relançar uma exceção, C++ passará a exceção para o bloco *try* mais externo. A razão mais provável pela qual você efetuará tal processamento dentro de seus programas será por você querer tratar uma exceção dentro de duas rotinas de tratamento separadas. Por exemplo, a rotina de tratamento dentro de uma função poderia tratar um aspecto da exceção, e a rotina de tratamento fora da função poderia tratar a outra. É importante compreender que relançar uma exceção passará *imediatamente* a exceção para a rotina de tratamento de fora — a rotina de tratamento interna não passará a exceção relançada. Para compreender melhor o processamento que C++ realiza quando você relança uma exceção, considere o seguinte programa, *duas_exc.cpp*, que lança uma exceção dentro de uma função, relança a exceção e pega-a novamente fora da função:

```
#include <iostream.h>

void TrataExcecao(void)
 {
   try {
      throw "Olá";
    }
   catch(char *) {
      cout << "Peguei char * dentro de TrataExcecao." << endl;
      throw;
    }
 }

void main(void)
 {
   cout << "Início: " << endl;
   try
      {
      TrataExcecao();
    }
   catch(char *)
   {
      cout << "Peguei char * dentro de main." << endl;
    }
   cout << "Fim";
 }
```

Quando você compilar e executar o programa *duas_exc.cpp*, ele chamará a função *TrataExcecao*. O bloco *try* interno pegará o comando *throw* dentro de *TrataExcecao* e gerará uma mensagem. O bloco *try* interno relançará a exceção, e o bloco *try* em *main* repegará a exceção. Quando você executar o programa *duas_exc.cpp*, sua tela exibirá a seguinte saída:

```
Início:
Peguei char * dentro de TrataExcecao.
Peguei char * dentro de main.
Fim
C:\>
```

APLICANDO O TRATAMENTO DA EXCEÇÃO 1140

Como você aprendeu, C++ fornece capacidades de tratamento de exceção para lhe oferecer uma estrutura que você possa usar para fazer seus programas responderem a eventos anormais ou não esperados. Como você usará *try* para lhe ajudar a tratar erros, tipicamente processará esses erros dentro de seus programas de um modo útil para seus programas (em vez das mensagens de erro simples que você gerou no bloco *catch* até aqui). Para compreender melhor o tratamento de exceção usando *try*, considere o seguinte programa, *catch_dz.cpp*, que gera e captura um erro de divisão por zero:

```
#include <iostream.h>

void divide(double a, double b)
 {
   try {
      if(!b) // verifica a divisão por zero
      throw b;
      cout << "Resultado: " << a/b << endl;
    }
   catch(double b)
      {
      cout << "Não é possível dividir por zero." << endl;
    }
 }
```

```
void main(void)
 {
   double i, j;
   do {
      cout << "Informe o numerador (0 para parar):" << endl;
      cin >> i;
      cout << "Informe o denominador: " << endl;
      cin >> j;
      divide (i,j);
    }
   while (i != 0);
 }
```

Compare o bloco try nesta dica com a rotina de tratamento de erro matemático projetada anteriormente na Dica 351. O bloco *try* é mais fácil de implementar, mais simples de compreender e mais eficiente que a rotina de erro matemático que você projetou antes. Como o tratamento da exceção fornece mais poder e é mais fácil de usar, à medida que seus programas se tornarem mais complexos, você o usará frequentemente para se proteger de eventos inesperados, em particular a entrada de dados do usuário.

1141 USANDO ARGUMENTOS PADRÃO DE FUNÇÃO

C++ lhe permite criar funções com valores padrão para os parâmetros da função. Suas funções somente usarão valores padrão para os parâmetros quando o comando de chamada não incluir explicitamente valores para seus parâmetros, ou quando o programa ainda não inicializou as variáveis que o comando de chamada usa para os parâmetros. Os valores padrão normalmente são úteis para seus programas, pois podem reduzir ou eliminar a necessidade de certas rotinas de tratamento de exceção.

No entanto, você geralmente usará valores de parâmetro padrão dentro de suas funções quando seu programa chamar repetidamente a função com o mesmo valor, e apenas ocasionalmente a chamará com um valor diferente. Por exemplo, o programa a seguir, *clr_wnew.cpp*, usa o caractere de nova linha para limpar a tela (como você aprendeu, usar o caractere de nova linha não é o modo mais eficiente de limpar a tela, mas é uma técnica útil para este exemplo). Como muitas telas têm 25 linhas de altura, a função *clrscr* assume por padrão 25 linhas:

```
#include <iostream.h>

void clrscr(int tamanho = 25)
 {
   while(tamanho > 0)
    {
      cout << endl;
      tamanho--;
    }
 }

void main(void)
 {
   int i;

   for(i = 0; i < 30; i++)
      cout << i << endl;
   cin.get();
   clrscr();                    // limpa 25 linhas
   for(i = 0; i < 30; i++)
      cout << i << endl;
   cin.get();
   clrscr(10);                  // limpa 10 linhas
 }
```

EVITANDO ERROS COM ARGUMENTOS DE FUNÇÃO PADRÃO 1142

Vimos que você pode freqüentemente usar argumentos padrão dentro de seus programas. Muitos programadores até mesmo usam argumentos padrão dentro das funções construtoras, como mostrado aqui:

```
class cubo {
   int x, y, z;
  public:
   cubo(int i = 0, int j = 0, int k = 0)
     {
       x = i;
       y = j;
       z = k;
     }
   // restante da definição
 }
```

Usar valores padrão dentro das funções construtoras pode lhe ajudar a evitar sobrecarregar as funções construtoras. No entanto, exatamente como com muitas das outras tecnologias simplificadoras que você aprendeu (tais como as funções genéricas), é preciso ter o cuidado de limitar o número de parâmetros para os quais você define valores padrão. Como regra geral, não deve declarar um valor padrão para um parâmetro a não ser que a função use o valor padrão 75% das vezes ou mais. Se sua função raramente usa o valor padrão, especificá-lo somente não é útil, mas também é destrutivo para seus programas e enganoso para os outros programadores que lerem seu código. Use valores padrão conforme necessário dentro de seus programas, mas tenha o cuidado de não abusar de seu uso.

ARGUMENTOS PADRÃO VERSUS SOBRECARGA DE FUNÇÃO 1143

Seus programas podem usar argumentos padrão para simplificar o processo de tratar os parâmetros de função. Como implica a dica anterior, você pode até mesmo usar argumentos padrão para evitar sobrecarregar funções, tais como as funções construtoras. Você também pode usar argumentos padrão como um método abreviado eficiente de fornecer funções sobrecarregadas. Imagine que você queira criar duas versões da sua própria função *concatstr*, que concatena duas strings. A primeira versão efetua seu processamento exatamente da mesma maneira que *strcat*, concatenando o conteúdo inteiro de uma string no final de outra. A segunda versão aceita um terceiro parâmetro que especifica o número de caracteres que a função deve concatenar (similar a *strncat*). Para sobrecarregar a função, você declararia os cabeçalhos para sua função *concatstr*, assim:

```
void concatstr(char *s1, char*s2);
void concatstr(char *s1, char *s2, int tam);
```

Alternativamente, você poderia usar um valor padrão para o parâmetro *tam*, que a função verificaria antes de iniciar seu processamento, assim:

```
void concatstr(char *s1, char *s2, int tam = 0);
 {
   if (tam = 0)
    {
      // processamento
    }
   else
    {
      // outro processamento
    }
 }
```

No exemplo anterior, usar um parâmetro padrão pode tornar seu programa mais fácil de compreender No entanto, como regra, tente usar qualquer forma de função que faça mais sentido e que torne seu processamento mais claro — para você e para outro programador que vier a ler seu código.

1144 CRIANDO FUNÇÕES DE CONVERSÃO

À medida que seus programas se tornarem mais complexos, algumas vezes você irá querer usar um objeto de uma classe em uma expressão que envolva outros tipos de dados. Como você aprendeu, pode freqüentemente usar as funções de operador sobrecarregado para lhe ajudar com esse processamento. No entanto, em outros casos, você pode simplesmente querer uma conversão de tipo do tipo de classe para o tipo do alvo. Para lhe ajudar com as conversões de tipo, C++ lhe permite criar *funções de conversão* personalizadas. Uma função de conversão converte sua classe em um tipo compatível com o resto de uma expressão. O formato generalizado de uma função de conversão de tipo é mostrado aqui:

```
operator tipo(void)   { return valor; }
```

No formato generalizado, *tipo* corresponde ao tipo alvo, e *valor* é o valor do objeto após a conversão. As funções de conversão não aceitam parâmetros e somente retornam um valor do tipo *tipo*. Para compreender melhor como as funções de conversão funcionam, considere o seguinte exemplo, que converte valores do tipo *pilha* para valores do tipo *int* retornando o valor *topo* (topo da pilha) para a expressão. O programa *conv_plh.cpp* usa a classe pilha que você criou nas dicas anteriores:

```
#include <iostream.h>

const int TAMANHO=100;

class pilha {
   int plh[TAMANHO];
   int topo;
 public:
   pilha(void)   { topo=0; }
   void push(int i);
   int pop(void);
   operator int(void)    { return topo; } // converte pilha para int
 };

void pilha::push(int i)
 {
   if(topo==TAMANHO)
    {
      cout << "A pilha está cheia." << endl;
      return;
    }
   plh[topo++] = i;
 }

int pilha::pop(void)
 {
   if(topo==0)
    {
      cout << "Erro na pilha." << endl;
      return 0;
    }
    return plh[--topo];
 }

void main(void)
 {
   pilha plh;
   int i, j;

   for(i = 0; i < 20; i++)
      plh.push(i);
   j = plh;           // converte para inteiro
```

```
   cout << j << " itens na pilha." << endl;
   cout << (TAMANHO - plh) << " espaços abertos." << endl;
}
```

USANDO FUNÇÕES DE CONVERSÃO PARA AUMENTAR A PORTABILIDADE DE TIPO 1145

Na dica anterior vimos que seus programas podem usar as funções de conversão para converter objetos de uma classe para outro tipo. Um dos melhores usos das funções de conversão é tornar seus tipos de objetos mais portáteis, e, portanto, mais úteis. Por exemplo, o programa a seguir, *pot_doub cpp*, usa a classe *potenc* que você criou anteriormente. No entanto, o programa *pot_doub cpp* também converte *potenc* para um *double* dentro de expressões, o que lhe permite usar o resultado de uma operação *potenc* dentro de outras equações matemáticas, como segue:

```
#include <iostream.h>

class potenc  {
   double b;
   int e;
   double val;
 public:
   potenc(double base, int exp);
   potenc operator+(potenc obj)
   {
      double base;
      int exp;
      base = b + obj.b;
      exp = e + obj.e;
      potenc temp(base, exp);
      return temp;
   }
   operator double(void)   { return val; } // converte para double
 };

potenc::potenc(double base, int exp)
 {
   b = base;
   e = exp;
   val = 1;
   if (exp !=0)
      while(exp   > 0)
         val *= b;
 }

void main(void)
 {
   potenc potenc1(4.0, 2);
   double doub1;

   doub1 = potenc1;            // converte x para double
   cout << (doub1 + 100.2) << endl;
   potenc potenc2(3.3, 3), potenc3(0,0);
   potenc3 = potenc1 + potenc2;   // nenhuma conversão
   doub1 = potenc3;               // converte para double
   cout << doub1;
 }
```

O programa *pot_doub.cpp* declara a variável *potenc1* (do tipo *potenc*), e a variável *doub1*, do tipo *double*. Ele então converte *potenc1* para um valor *double*, e exibe o valor convertido. Em seguida, o programa declara mais

duas variáveis do tipo *potenc*, *potenc2* e *potenc3*, com as quais realiza processamento e depois exibe. Quando você compilar e executar o programa *pot_doub.cpp*, sua tela exibirá a seguinte saída:

```
116.2
20730.7
C:\>
```

1146 FUNÇÕES DE CONVERSÃO VERSUS OPERADORES SOBRECARREGADOS

Como aprendeu, você poderia realizar essencialmente o mesmo processamento que realizou nas Dicas 1144 e 1145 sobrecarregando operadores relativos às classes *pilh* e *potenc*, respectivamente. Porém, como também aprendeu, você precisa sobrecarregar os operadores de forma diferente para que seus programas executem cada uma das duas seguintes atribuições:

```
x = potenc + 102.65;
x = 102.65 + potenc;
```

No entanto, em vez de usar operadores sobrecarregados, você pode criar uma função de conversão, como detalhou a dica anterior. Uma função de conversão lhe ajudaria a evitar sobrecarregar múltiplos operadores, usando funções *amigas*, ou efetuando outro processamento complexo e repetitivo para converter um objeto para outro valor. Por outro lado, se você estivesse trabalhando com uma classe e quisesse adicionar *102.65* a cada um de seus membros, a função de conversão não lhe ajudaria. Exatamente como você viu com parâmetros padrão e funções sobrecarregadas, a decisão de usar uma função de conversão *versus* um operador sobrecarregado variará de programa para programa e de classe para classe. Exatamente como nos exemplos anteriores, você deverá tomar suas decisões com base no aplicativo específico da sua classe dentro de seu programa.

1147 COMPREENDENDO OS NOVOS OPERADORES DE CONVERSÃO DE C++

Você aprendeu nas dicas anteriores que C suporta um operador de conversão, que seus programas podem usar para converter um valor para um tipo alvo. C++ define quatro novos operadores de conversão, listados na Tabela 1147.

Tabela 1147 Os novos operadores de conversão suportados por C++.

Nome	Forma Geral
const_cast	const_cast<tipo>(objeto)
dynamic_cast	dynamic_cast<tipo>(objeto)
reinterpret_cast	reinterpret_cast<tipo>(objeto)
static_cast	static_cast<tipo>(objeto)

Seus programas podem continuar usando os operadores de conversão de C e devem também usar os novos operadores de conversão de C++, como apropriado para o processamento do programa. Você aprenderá mais sobre cada novo operador de conversão nas dicas a seguir.

1148 USANDO O OPERADOR CONST_CAST

C++ define vários novos operadores de conversão que podem ser usados dentro de seus programas. Você usará o operador *const_cast* para anular explicitamente uma declaração *const* ou *volatile* anterior. O tipo alvo da conversão precisa ser igual ao tipo de origem, exceto para a alteração de seu atributo *const* ou *volatile*. Você freqüentemente usará *const_cast* para remover o atributo constante de um valor — em outras palavras, para tornar um valor que você definiu anteriormente como constante em modificável.

Seu programa pode usar o operador *const_cast* para converter explicitamente um ponteiro para qualquer tipo de objeto ou um ponteiro para um membro de dados para um tipo que é idêntico exceto para os qualificadores *const, volatile* e *_unaligned*. Para ponteiros e referências, o resultado referenciará o objeto original. Para ponteiros para dados-membro, o resultado referenciará o mesmo membro que o ponteiro original (não-convertido) para um dado-membro. Por exemplo, o programa a seguir, *const.cpp*, não passa na compilação porque você não pode atribuir um valor ponteiro constante para um valor ponteiro normal:

```
#include <stdio.h>

class c  {
 public:
   int j;
   c(void)  (j = 10;)
 };

void ImprimeInt( const c* Objeto)
 {
   c* New = Objeto;
   New->j += 5;
   printf("%d\n", New->j);
 }

void main(void)
 {
   const c Exemplo;

   ImprimeInt(&Exemplo);
 }
```

No entanto, se você alterar a função *ImprimeInt*, para usar o operador *const_cast*, o programa compilará e executará corretamente, como mostrado aqui em um fragmento de código do programa *const_cast.cpp:*

```
void ImprimeInt(const c* Objeto)
 {
   c* New = const_cast<c*> (Objeto);
   New->j += 5;
   printf("%d\n", New->j);
 }
```

A conversão lhe permite criar o ponteiro *New* como um ponteiro modificável. No entanto, você deve ser cuidadoso com *const_cast*. Dependendo do tipo do objeto referenciado, uma operação de escrita por meio do ponteiro resultante, referência ou ponteiro para um dado-membro poderá produzir comportamento desconhecido ou inesperado. O fragmento de código a seguir resultará em um erro:

```
class X {};
class Y : public X {};

const X x;
Y y = const_cast<Y> (x);     // erro
```

O código resulta em um erro porque você não pode usar o operador *const_cast* para converter de um X para um Y. Você somente pode usar o operador *const_cast* para remover os modificadores *const, volatile* e *_unaligned*. Se você quisesse converter o objeto constante *x* para um objeto não-constante *y*, precisaria construir a operação de conversão como segue:

```
Y y = (const_cast<Y>)(static_cast<const Y>(x));
```

Nota: O operador const_cast converte um valor ponteiro NULL para o valor de ponteiro NULL do tipo de destino.

1149 USANDO O OPERADOR DYNAMIC_CAST

Como você aprendeu, C++ fornece vários operadores novos de conversão, que você pode usar dentro de seus programas. A operação *dynamic_cast* efetua uma conversão na execução e verifica a validade da conversão. Se o programa não puder fazer a conversão, a conversão falhará e a expressão será avaliada como *NULL*. Geralmente, você usará o operador *dynamic_cast* para efetuar conversões em tipos de objeto polimórficos. Por exemplo, *dynamic_cast* pode retornar um ponteiro para um objeto derivado dado um ponteiro para a classe-base polimórfica. Para compreender melhor o processamento que o operador *dynamic_cast* realiza, considere o seguinte programa de exemplo, *dyn_cast.cpp*, que gera uma classe a partir de duas classes-base e tenta converter ponteiros das classes-base para a classe derivada:

```
// NOTA: Compile este programa com a opção Generate RTTI
// do seu compilador habilitada.

#include <iostream.h>
#include <typeinfo.h>

class Base1
 {
   virtual void f(void)  {  /* Uma função virtual torna a
                               classe polimórfica  */ }
 };

class Base2 { };
class Derivada : public Base1, public Base2 { };

void main(void)
 {
   try
   {
      Derivada d, *pd;
      Base1 *b1 = &d;

      // Efetua uma conversão de uma Base1 para uma Derivada.
      if ((pd = dynamic_cast<Derivada *>(b1)) != 0)
        {
          cout << "O ponteiro resultante é do tipo " << typeid(pd).name()
           << endl;
        }
      else
      throw Ma_conversao();

      // Converte da primeira base para a classe derivada
      // e depois volta novamente para outra base acessível
      Base2 *b2;
      if ((b2 = dynamic_cast<Base2 *>(b1)) != 0)
        {
          cout << "O ponteiro resultante é do tipo " << typeid(b2).name()
           << endl;
        }
      else
      throw Ma_conversao();
      }
   catch (Ma_conversao)
   {
      cout << "dynamic_cast falhou" << endl;
      exit(1);
      }
   catch (...)
    {
```

```
        cout << "Erro de tratamento de exceção." << endl;
        exit(1);
        }
}
```

Quando você executar o programa *dyn_cast.cpp*, ele efetuará duas tentativas de conversão. Primeiro, tentará converter de um ponteiro *Base1* para um ponteiro *Derivada*. Em seguida, converterá da primeira classe-base para a classe mais derivada e depois tentará converter a árvore para outra classe-base. Quando você compilar e executar o programa *dyn_cast.cpp*, sua tela exibirá a seguinte saída:

```
O ponteiro resultante é do tipo Derivada *
O ponteiro resultante é do tipo Base2 *
C:\>
```

Nota: Você precisa compilar o programa dyn_cast.cpp com a opção Generate RTTI do seu compilador habilitada, ou o programa não poderá efetuar identificação de tipo durante a execução.

USANDO O OPERADOR REINTERPRET_CAST 1150

C++ fornece vários operadores novos de conversão, que você pode usar dentro de seus programas. O operador *reinterpret_cast* converte um tipo para um tipo incompatível e fundamentalmente diferente. Por exemplo, o operador *reinterpret_cast* poderia converter um ponteiro do tipo *potenc* para um ponteiro do tipo *int*. Claramente, *reinterpret_cast* apresenta a oportunidade para muita confusão dentro de seus programas, e você deve evitá-lo a não ser que seu uso seja absolutamente necessário. O programa a seguir, *ren_cast.cpp*, usa o operador *reinterpret_cast* para converter um ponteiro char para um ponteiro *int*:

```
#include <iostream.h>

void main(void)
 {
   int i;
   char *p = "Isto é uma string.";

   i = reinterpret_cast<int> (p);  // converte ponteiro char para inteiro
   cout << i;
 }
```

Quando você compilar e executar o programa *ren_cast.cpp*, ele exibirá saída sem sentido porque a conversão do ponteiro *char* para um ponteiro *int* faz o ponteiro *int* retornar resultados incomuns. Se você rodar o programa em um sistema de 16 bits, *int* retornará *char('I')*. Se, por outro lado, você rodar o programa em um sistema de 32 bits com suporte para inteiros de 32 bits, o ponteiro *i* retornará um equivalente bit a bit da constante string 'Is'. Em um sistema de 32 bits, a saída fica similar ao seguinte:

```
4247824
C:\>
```

USANDO O OPERADOR STATIC_CAST 1151

Como você aprendeu, C++ fornece vários operadores novos de conversão. Seus programas usarão o operador *static_cast* para efetuar uma conversão não-polimórfica. Em outras palavras, você pode usar o operador *static_cast* para converter um ponteiro de classe-base para um ponteiro de classe derivada. Você usará o operador *static_cast* dentro de seus programas como mostrado no seguinte protótipo:

```
static_cast< T > (argumento)
```

Na forma generalizada mostrada aqui, *T* é um ponteiro, referência, tipo aritmético ou tipo enumerado. O tipo *argumento* precisa corresponder ao tipo *T*. O compilador precisa conhecer tanto *T* quanto *argumento* durante a compilação. Se seu programa puder converter um tipo completo para outro tipo por algum método de conversão que a linguagem já forneça, fazer essa conversão usando o operador *static_cast* produzirá a mesma conversão. Adicionalmente, seus programas podem usar *static_cast* para converter tipos inteiros para tipos enume-

rados. Uma solicitação para converter *argumento* para um valor que não seja um elemento de *enum* retorna um valor de *undefined*. O ponteiro *NULL converte uma coisa em uma coisa* (em outras palavras, um *static_cast* de *NULL* ainda produz *NULL*), e, portanto, seus programas não precisam usar *static_cast* com o ponteiro *NULL*.

Seu programa pode converter um ponteiro para um tipo de objeto em um ponteiro para outro tipo de objeto. Observe que meramente apontar para tipos similares poderá causar problemas de acesso se você não alinhar os tipos similares de forma correspondente. Você pode converter explicitamente um ponteiro para uma classe base para um ponteiro para alguma outra classe *derivada*, se *base* for uma classe-base para *derivada*. Seu programa pode fazer uma conversão estática somente nas seguintes condições.

- Se uma conversão não-ambígua existir de *derivada* para *base*
- Se *base* não for uma classe-base virtual

O operador *static_cast* pode converter explicitamente um objeto em um tipo de referência *base&* se o compilador puder converter explicitamente um ponteiro para esse objeto no tipo ponteiro *base**. O resultado da conversão *static_cast* é um *lvalue*. Seu programa não chama funções construtoras ou de conversão como resultado de uma conversão para uma referência. Em vez disso, seu programa pode converter um objeto ou um valor para um objeto de classe somente se você tiver declarado uma construtora apropriada ou operador de conversão para essa conversão específica. Você pode converter explicitamente um ponteiro para um membro em um tipo ponteiro para membro diferente somente se ambos os tipos forem ponteiros para membros da mesma classe ou ponteiros para membros de duas classes. Se tanto o argumento e o resultado forem ponteiros para membros de duas classes, seu programa precisará derivar a classe que um ponteiro referencia não ambiguamente a partir de outra classe.

Quando *T* for uma referência, o resultado de um *static_cast* será um *lvalue*. O resultado de uma conversão de ponteiro ou de referência refere-se à expressão original.

1152 COMPREENDENDO OS NAMESPACES

À medida que seus programas se tornarem mais complexos, seus aplicativos eventualmente consistirão de mais de um arquivo-fonte. Além disso, mais de um desenvolvedor pode ser o autor e fazer manutenções em cada arquivo-fonte. De forma eventual, você organizará e vinculará os arquivos separados para produzir o aplicativo final. Tradicionalmente, a organização de arquivos requer que todos os nomes que um arquivo-fonte não encapsule dentro de um *namespace* definido (em outras palavras, nomes que não tenham um escopo limitado, tal como um corpo de função, corpo de classe, ou unidade de tradução) necessitem compartilhar o mesmo *namespace* global. Portanto, múltiplas definições de nomes que o compilador descobre ao vincular módulos separados requerem algum modo de distinguir cada nome. A palavra-chave *namespace* de C++ fornece a solução para o problema de "conflito de nomes" no escopo global.

A palavra-chave *namespace* lhe permite particionar um aplicativo em múltiplos subsistemas. Cada subsistema pode definir e operar dentro de seu próprio escopo. Cada desenvolvedor pode introduzir identificadores convenientes em um subsistema sem se preocupar em saber se alguma outra pessoa também está usando esses identificadores dentro de seus próprios subsistemas. Cada namespace usa um identificador único. Quando seus programas definem namespaces, o compilador conhece o escopo do subsistema em todo o aplicativo pelo identificador único de cada namespace.

Usar namespaces de C++ requer somente duas etapas. A primeira é usar a palavra-chave *namespace* para identificar de forma exclusiva um namespace. A segunda etapa é usar a palavra chave *using* para acessar os elementos de um namespace identificado anteriormente.

1153 USANDO NAMESPACES

Como você aprendeu na dica anterior, as questões de escopo e de nomeação tornam-se mais importantes à medida que seus programas forem se tornando mais complexos. Para lhe ajudar a evitar conflitos de variáveis e de outros nomes, C++ fornece a palavra-chave namespace. Você usará a palavra-chave namespace dentro de seus programas de forma muito parecida à forma como usa uma estrutura, um tipo enumerado, uma união ou uma definição de classe, como mostrado aqui na forma generalizada da implementação de *namespace:*

```
namespace name  {
    // declarações de objeto
}
```

Dentro de seus próprios programas, você pode incluir variáveis de qualquer tipo que o compilador já conheça (ou tipos de C ou tipos de C++ ou de classes anteriormente definidas, estruturas e uniões) dentro da definição de *namespace*. Você também pode declarar funções *in-line* dentro de um *namespace*. Por exemplo, o seguinte fragmento de código declara duas variáveis e uma função dentro do namespace *limitado*:

```
namespace limitado   {
   int i, k;
   void exemplo(int j)  { cout << j  << endl;  }
 }
```

No fragmento de código anterior, *i*, *k* e a função *exemplo* fazem parte do namespace *limitado*. Como um namespace define um escopo, você precisa usar o operador de resolução de escopo para referenciar os objetos que definir dentro de um namespace. Por exemplo, para atribuir o valor 10 à variável *k*, você precisa usar um comando similar ao seguinte:

```
limitado::k = 10;
```

USANDO O COMANDO USING COM NAMESPACE 1154

Na dica anterior você aprendeu que, quando seus programas usam namespaces, você precisa resolver referências para objetos dentro de um namespace com o operador de resolução de escopo No entanto, se seu programa freqüentemente usar os membros de um namespace, você poderá usar um comando *using* para simplificar o acesso do programa para esses membros. O comando *using* tem duas formas gerais, como segue:

```
using namespace, nome;
using nome::membro;
```

A primeira forma lhe permite acessar o namespace inteiro. Na segunda forma, você define somente membros específicos do namespace que quer acessar. Essencialmente, a primeira forma torna o namespace inteiro público, e a segunda lhe permite encapsular certos membros dentro do namespace. Para compreender melhor as duas formas do comando *using*, considere o seguinte fragmento de código, que usa ambas as formas:

```
using limitado::k;     // somente torna k visível
k = 10;

using namespace limitado;  // torna todo o namespace limitado visível
k = 10;
```

COMPREENDENDO A IDENTIFICAÇÃO DE TIPO EM TEMPO DE EXECUÇÃO 1155

Uma adição importante nos novos compiladores C++ é a identificação de tipo em tempo de execução (RTTI, de run-time type identification), que lhe permite escrever o código portátil que pode determinar o tipo real de um objeto de dados em tempo de execução mesmo quando o código pode somente acessar um ponteiro ou uma referência a esse objeto. A identificação de tipo em tempo de execução torna possível, por exemplo, converter um ponteiro para uma classe-base virtual em um ponteiro para o tipo derivado do objeto real. Como foi visto na Dica 1149, você pode usar o operador *dynamic_cast* junto com a identificação de tipo em tempo de execução para fazer conversões em tempo de execução.

USANDO TYPEID PARA A IDENTIFICAÇÃO DE TIPO EM TEMPO DE EXECUÇÃO 1156

Na dica anterior vimos que a identificação de tipo em tempo de execução permite que seus programas manipulem ponteiros e referências de formas totalmente novas. O mecanismo de identificação de tipo na execução também lhe permite verificar se um objeto é de algum tipo particular e se dois objetos são do mesmo tipo. Você pode verificar os objetos com o operador *typeid*. O operador *typeid* determina o tipo real de seu argumento e retorna uma referência a um objeto do tipo *const typeinfo*, que descreve esse tipo.

Você também poderá usar um nome de tipo como o argumento de *typeid*, e *typeid* retornará uma referência a um objeto *const typeinfo* para esse tipo. A classe *typeinfo* fornece um operador == e um operador != que você pode usar para determinar se dois objetos são do mesmo tipo. A classe *typeinfo* também fornece uma função-membro, *name*, que retorna um ponteiro para uma string de caracteres que contém o nome do tipo. Você precisa incluir o arquivo de cabeçalho *typeinfo.h* dentro de seus programas para acessar a função *typeid*. A forma geral da função *typeid* é mostrada aqui:

```
#include <typeid.h>
const typeinfo typeid(objeto);
```

Dentro da forma generalizada, *objeto* corresponde ao objeto cujo tipo você quer que *typeid* retorne. Quando você aplicar *typeid* a um ponteiro de classe-base de uma classe polimórfica, *typeid* automaticamente retornará o tipo do objeto para o qual o ponteiro aponta, incluindo quaisquer classes derivadas da classe-base. Para compreender melhor o processamento que *typeid* realiza, considere o seguinte programa, *typeid_1.cpp*:

```
#include <iostream.h>
#include <typeinfo.h>

class A { };
class B : A { };

void main(void)
 {
   char C;
   float X;

   //USA O typeinfo::operator == () PARA FAZER COMPARAÇÃO
   if (typeid( C ) == typeid( X ))
      cout << "C e X são do mesmo tipo." << endl;
   else
      cout << "C e X NÃO são do mesmo tipo." << endl;

   // USA AS LITERAIS true E false PARA FAZER COMPARAÇÃO
   cout << typeid(int).name();
   cout << " antes " << typeid(double).name() << ": " <<
    (typeid(int).before(typeid(double)) ? true : false) << endl;
   cout << typeid(double).name();
   cout << " antes " << typeid(int).name() << ": " <<
           (typeid(double).before(typeid(int)) ? true :
           false) << endl;
   cout << typeid(A).name();
   cout << " antes " << typeid(B).name() << ": " << (typeid(A).before
    (typeid(B)) ? true : false)
        << endl;
 }
```

O programa *typeid_1.cpp* declara duas classes, uma classe-base (A) e uma classe derivada (B). Quando *typeid_1.cpp* inicia sua execução, ela define duas variáveis, uma do tipo *char* e uma do tipo *float*. O programa em seguida testa os tipos de *X* e de *C*. Se elas forem do mesmo tipo, o programa exibirá uma mensagem dizendo isso na tela; caso contrário, o programa exibirá uma mensagem dizendo que elas não são do mesmo tipo. Claramente, como *char* e *float* não são do mesmo tipo, o programa exibirá a mensagem "C e X NÃO são do mesmo tipo". O programa em seguida usa o membro *before* para comparar alguns tipos básicos com uma comparação léxica. Uma comparação léxica é uma comparação baseada no alfabeto: a vem antes de b, c antes de d e após b, e assim por diante. Portanto, quando o programa compara *int* e *double*, ele retorna *false* quando testa se *double* está após *int*, mas *true* quando testa se *double* está antes de *int*. Finalmente, o programa testa se A está antes de B. Quando você compilar e executar o programa *typeid_1.cpp*, ele exibirá a seguinte saída na tela:

```
C e X NÃO são do mesmo tipo.
int antes de double: 0
double antes de int: 1
A antes de B: 1
C:\>
```

COMPREENDENDO A CLASSE TYPE_INFO 1157

Vimos na dica anterior que a função *typeid* retorna um valor do tipo *const typeinfo*. O valor *const typeinfo* é uma string que representa informações de tipo sobre a classe. A Tabela 1157 lista os possíveis valores de retorno para a função *typeid*.

Tabela 1157 Os valores possíveis que *typeid* retorna.

Valor	Significado
[ARRAY]	O valor é uma matriz. A função *typeid* sempre retorna [ARRAY] junto com outra palavra-chave.
nomeclasse	O nome da classe definida pelo usuário para o objeto.
[INTEGER]	O objeto é um inteiro ou inteiro longo.
[NULL]	O objeto é um valor *NULL* (geralmente um ponteiro *NULL*).
[REAL]	O objeto representa um objeto *float*, *double* ou *long double*.
[STRING]	O objeto é uma string de caracteres.
[UNINITIALIZED]	O objeto não está inicializado (geralmente um ponteiro para uma classe polimórfica).

A classe *typeinfo* também define quatro membros públicos, além da função *typeinfo*. Você implementará essas funções-membro dentro de seus programas, como mostrado aqui:

```
bool operator==(const type_info &obj) const;
bool operator!=(const type_info &obj) const;
bool before(const type_info &obj) const;
const char *name(void) const;
```

Como você aprendeu na dica anterior, os operadores sobrecarregados == e != lhe permitem comparar os tipos que *typeid* retorna. O compilador usa principalmente a função *before* internamente. Ela retornará *true* se o objeto chamador estiver antes do objeto *obj* na ordem léxica (uma lista derivada do compilador). Ela não retorna informações com relação às hierarquias de classes ou outras informações úteis sobre o tipo. A função *name* retorna um ponteiro para o nome do tipo. O programa a seguir, *typeid_2.cpp*, usa a função *name* para retornar informações adicionais a partir de uma ação *typeid*:

```
#include <strings.h>
#include <iostream.h>
#include <typeinfo.h>

class Base {
   int a, b;
   virtual void func(void) {};
 };

class Derivada1: public Base {
   int i, j;
 };

class Derivada2: public Base {
   int k;
 };

void main(void)
 {
   int i;
   Base *p, baseobj;
   Derivada1 obj1;
   Derivada2 obj2;

   cout << "Typeid de i é: ";
   cout << typeid(i).name() << endl;
```

```
    p = &baseobj;
    cout << "p está apontando atualmente para um objeto do tipo: ";
    cout << typeid(*p).name() << endl;
    p = & obj1;
    cout << "p agora está apontando para um objeto do tipo: ";
    cout << typeid(*p).name() << endl;
    p = & obj2;
    cout << "p está finalmente apontando para um objeto do tipo: ";
    cout << typeid(*p).name() << endl;
}
```

O programa *typeid_2.cpp* define uma única classe-base e duas classes derivadas. Quando o programa inicia sua execução, ele declara duas variáveis dos tipos derivados, uma variável do tipo base, um ponteiro para o tipo base e uma variável inteira simples.

Em seguida, o programa verifica o *typeid* da variável inteira e retorna seu nome no texto. Após retornar informações sobre a variável de tipo *int* simples, o programa manipula *p* para apontar para cada uma das três classes personalizadas. Cada vez que o tipo *p* aponta para mudanças, o programa exibe informações sobre o novo nome de tipo. Quando você compilar e executar o programa *typeid_2.cpp*, ele exibirá a seguinte saída:

```
Typeid de i é: int
p está apontando atualmente para um objeto do tipo: Base
p agora está apontando para um objeto do tipo: Derivada1
p está finalmente apontando para um objeto do tipo: Derivada2
C:\>
```

1158 COMPREENDENDO A PALAVRA-CHAVE MUTABLE

Como você aprendeu, C++ acrescenta novos especificadores às declarações de variáveis, tais como o especificador *long double*, que você pode usar dentro de seus programas. Além dos novos tipos simples de dados, seus programas podem usar a palavra-chave *mutable* com uma variável de qualquer tipo para tornar a variável modificável — embora ela esteja em uma expressão *const* qualificada. Por exemplo, a seguinte declaração torna *j* uma variável *mutable int:*

```
class ExibeExemplo {
   mutable int j;
};
```

Seus programas podem declarar somente membros da classe de dados como *mutable*. Você não pode usar a palavra-chave *mutable* com nomes *static* ou *const*. O propósito de *mutable* é especificar quais membros de dados as funções-membro *const* podem modificar, porque uma função-membro *const* normalmente não pode modificar dados-membro.

1159 USANDO A PALAVRA-CHAVE MUTABLE DENTRO DE UMA CLASSE

A dica anterior mostrou que, seus programas podem usar a palavra-chave *mutable* dentro de uma definição de classe para tornar uma variável membro modificável, embora ela esteja em uma expressão qualificada por *const*, mesmo quando o objeto do qual ela é um membro for *const*. Você somente pode usar *mutable* em membros dentro de uma classe *const*. Para compreender melhor o uso da palavra-chave *mutable*, considere o programa a seguir, *mutable.cpp*, que declara dois membros *mutable* dentro de uma classe:

```
#include <iostream.h>
class Alfa    {
   mutable int conta;
   mutable const int* iptr;
public:
   Alfa(void)    {conta = 0;}
   int func1(int i = 0) const  {  // Prometa não alterar os argumentos const.
      conta = i++;  // Mas conta pode ser modificado.
```

```
      iptr = &i;
      cout << "i é: " << *iptr << endl;
      return conta;
      }
    void exibe_conta(void)  { cout << "Conta é: " << conta << endl; }
  };

void main(void)
  {
    Alfa a;

    a.exibe_conta();
    a.func1(10);
    a.exibe_conta();
  }
```

Em vez de deixar o membro *conta* não-modificado, como promete o modificador *const* da declaração *func1*, a palavra-chave *mutable* permite que a função-membro modifique o membro *conta*. Em vez de manter seu valor inicial em 0, *conta* tem um valor de 10 no final do programa porque *conta* é um valor *mutable*. Quando você compilar e executar o programa *mutable.cpp*, sua tela exibirá a seguinte saída:

```
Conta é: 0
i é: 11
Conta é: 10
C:\>
```

CONSIDERAÇÕES SOBRE A PALAVRA-CHAVE MUTABLE 1160

Como você aprendeu, seus programas podem usar a palavra-chave *mutable* para anular o estado de constante aplicado a um membro de uma classe. Embora a palavra-chave *mutable* lhe ofereça meios para controlar melhor quais partes de uma classe devem permanecer não-modificadas e quais partes devem ser modificáveis, a palavra-chave *mutable* também pode introduzir erros significativos e difíceis de seguir. Por exemplo, no programa *mutable.cpp*, apresentado na dica anterior, a classe *Alfa* declarou a variável *conta* como *mutable*. Mais tarde, a classe definiu a função *const*, que não deve ter alterado os valores de quaisquer variáveis que a função usou. Em vez disso, como a classe declarou a variável *conta* com a palavra-chave *mutable*, a função *func1* alterou o valor do membro *a.conta* para 10.

Você precisa ser muito cuidadoso sobre como e quando usar a palavra-chave *mutable* dentro de seus programas, pois fazer isso poderá criar erros difíceis de seguir. Como com muitos recursos que C++ acrescenta, sua decisão de usar ou não *mutable* geralmente dependerá da clareza. O programa *mutable.cpp* da dica anterior ilustra a importância da clareza em que é muito menos claro com as palavras-chave *mutable* do que seria em caso contrário.

INTRODUZINDO O TIPO DE DADOS BOOL 1161

Em dicas anteriores, você usou o tipo de dados *int* dentro de suas comparações lógicas. Os compiladores C++ mais recentes suportam o tipo de dados *bool* para o processamento de dados Booleanos. O tipo de dados *bool* aceita valores *true* e *false*. Você declarará e usará variáveis do tipo de dados *bool* como mostrado aqui:

```
bool var_logica;

var_logica = true;
```

Como o tipo de dados *bool* suporta as palavras-chave *true* e *false* como *rvalues*, você não deve definir variáveis chamadas *true* ou *false* dentro de seus programas.

USANDO O TIPO DE DADOS BOOL 1162

Na dica anterior você viu que C++ suporta o tipo de dados *bool*, que seus programas podem usar para manter informações lógicas. O novo tipo de dados *bool* usa duas novas palavras-chave de C++: *true* e *false*. As palavras-

chave *true* e *false* correspondem aos valores verdadeiro (1) e falso (0) para inteiros. Você deve usar o tipo de dados *bool* dentro de seus programas para tornar o código do seu programa mais claro. Por exemplo, você pode atribuir o resultado de um teste lógico para um tipo de dados *bool*, como segue:

```
bool result;

result = (A && B)
```

O programa a seguir, *bool_fun.cpp*, usa uma função que retorna um valor do tipo *bool*:

```
#include <iostream.h>

bool func(void)
 {                      // A função retorna um tipo bool
   return false;
   //  return NULL;   // NULL é convertido para Booleno false
 }

void main(void)
 {
   bool val = false;  // Variável Booleana
   int i = 1;         // i não é nem o booleano True
                      // nem o booleano False
   int g = 3;
   int *iptr = 0;     // ponteiro null
   float j = 1.01;    // j não é nem Booleano True nem o
                      // Booleano False

   if (i == true)
      cout << "True: valor é 1" << endl;
   if (i == false)
      cout << "False: valor é 0" << endl;
   if (g)
      cout << "g é true.";
   else
      cout << "g é false.";

   // Teste no ponteiro
   if (*iptr == false)
      cout << "Ponteiro inválido." << endl;
   if (*iptr == true)
      cout << "Ponteiro válido." << endl;

   // Para testar o valor verdadeiro de j, converta-
   // o para o tipo bool.
   if (bool(j) == true)
      cout << "O booleano j é True." << endl;

   // Testa o valor de retorno da função booleana

   val = func();
   if (val == false)
      cout << "func() retornou false.";
   if (val == true)
      cout << "func() retornou true.";
 }
```

Além de usar uma função *bool*, o programa executa uma série de comparações usando valores *bool* em vez de inteiros, e converte um valor *int* para um resultado *bool*. Quando você compilar e executar o programa *bool_fun.cpp*, sua tela exibirá a seguinte saída:

```
True: valor é 1
```

```
Ponteiro inválido.
O booleano j é True.
func() retornou False.
C:\>
```

CRIANDO UM TIPO STRING 1163

Como você aprendeu, C++ implementa strings como matrizes de caracteres terminadas por *NULL* e não como um tipo de dado separado. A maioria das versões atuais de C++ implementa a biblioteca padrão *<string.h>*, que cria o tipo de dados separado para strings. Para compreender melhor como implementar e manipular classes inteiras dentro de seus programas, você aprenderá a criar um tipo string personalizado. Nas próximas dicas, você usará o que aprendeu nas dicas anteriores para criar um tipo de dados string completo.

A primeira etapa na criação de qualquer novo tipo é determinar o que você quer realizar. Em resumo, você precisa *definir* o tipo. Na dica a seguir, você definirá as características do seu tipo string personalizado — os operadores que seus programas podem usar com o tipo, as funções-membro que o tipo suporta e assim por diante.

DEFININDO AS CARACTERÍSTICAS DO TIPO STRING 1164

Como você sabe, ao trabalhar com uma matriz *char* em C++, você está, na verdade, trabalhando com vários elementos *char* que C++ armazena em seqüência dentro da memória do computador. Por exemplo, a definição mostrada aqui na verdade cria 32 elementos *char* separados dentro da matriz *amostra*.

```
char amostra[] = "Bíblia do Programador C/C++";
```

Em geral, seria significativamente mais fácil atribuir o valor direto à string *amostra*, como mostrado aqui:

```
Strings amostra = "Bíblia do Programador C/C++";
```

Se você criar o tipo do modo mostrado no segundo exemplo, poderá mais tarde atribuir a string à outra string, como segue:

```
Strings amostra1, amostra2;
amostra1 = "Bíblia do Programador C/C++";
amostra2 = amostra1;
```

Claramente, o fragmento de código anterior, que usa *Strings*, é mais fácil de compreender do que o seguinte:

```
Char amostra[] = "Bíblia do Programador C/C++";
Char amostra2[256];
strcpy(amostra2, amostra);
```

Adicionalmente, sua classe *Strings* deve lhe permitir usar o operador +, em vez da função *strcat* que você aprendeu nas dicas anteriores, para concatenar duas strings juntas, como mostrado aqui:

```
Strings amostra, amostra2;
amostra = "Bíblia do Programador";
amostra2 = "C/C++, do Jamsa!";
amostra = amostra + amostra2;
```

Sua classe *Strings* também deve lhe permitir acrescentar uma string de caracteres tipada não-*Strings* a uma string, como mostrado aqui:

```
amostra = amostra + "Isto é um exemplo";
```

Similarmente, sua classe *Strings* deve lhe permitir remover uma substring a partir de dentro de uma string usando o operador de subtração. Finalmente, sua classe *Strings* deve lhe permitir comparar strings usando operadores relacionais, tais como os operadores maior e menor, em vez da função *strcmp*. Em outras palavras, o fragmento de código a seguir deve comparar precisamente *amostra1* e *amostra2*:

```
Strings amostra1, amostra2;
amostra1 = "Bíblia do Programador C/C++";
amostra2 = "Bíblia do Programador C/C";
```

```
if(amostra2 > amostra1)
    cout << amostra2 << " é maior que " << amostra1;
```

Agora que você compreende algumas operações básicas que quer que sua classe *string* realize, você deve começar a implementar a classe *Strings*. Na dica a seguir, você criará a declaração para sua classe *Strings*.

1165 CRIANDO A CLASSE STRINGS

Já sabemos que você, pode usar uma classe de C++ para criar seu próprio tipo string, em vez de continuar a usar as matrizes *char*. Na dica anterior, você revisou algumas operações básicas que pode querer que sua classe string execute. Na verdade, a classe *Strings* que você definirá nesta dica é construída inteiramente a partir da definição que você criou para sua classe *Strings* na dica anterior. Você deve observar que a definição da sua classe *Strings* usa funções *friend* para sobrecarregar apenas alguns operadores, e usa principalmente funções-membro para suas sobrecargas de operadores relacionais. No entanto, você também poderia escrever a classe usando funções *friend* e ainda efetuar o mesmo processamento que a definição atual. O CD-ROM que acompanha este livro inclui o arquivo *strings.cpp*, que contém a definição de classe para a classe *Strings*, como mostrado aqui:

```
class Strings {
   char *p;
   int tamanho;
 public:
   Strings(char *str);
   Strings(void);
   Strings(const Strings &obj);  // Construtora da cópia
   ~Strings(void) {delete [] p;}

   friend ostream &operator<<(ostream &stream, Strings &obj);
   friend istream &operator>>(istream &stream, Strings &obj);

   Strings operator=(Strings &obj); // atribui um objeto Strings
   Strings operator=(char *s); // atribui uma string entre aspas
   Strings operator+(Strings &obj); // concatena um objeto Strings
   Strings operator+(char *s); // concatena uma string entre aspas
   friend Strings operator+(char *s, Strings &obj);
            /* concatena uma string entre aspas com um
               objeto Strings */

   Strings operator-(Strings &obj); // subtrai um objeto Strings
   Strings operator-(char *s);  // subtrai uma string entre aspas

 /* operadores relacionais entre objetos Strings. Observe
    que os operadores poderiam facilmente retornar bool,
    em vez de int */

   int operator==(Strings &obj) {return !strcmp(p, obj.p);}
   int operator!=(Strings &obj) {return strcmp(p, obj.p);}
   int operator<(Strings &obj) {return strcmp(p, obj.p) < 0;}
   int operator>(Strings &obj) {return strcmp(p, obj.p) > 0;}
   int operator<=(Strings &obj) {return strcmp(p, obj.p) <= 0;}
   int operator>=(Strings &obj) {return strcmp(p, obj.p) >= 0;}

 /* operadores relacionais entre objeto Strings e uma
    string de caracteres delimitada.
    Observe que os operadores poderiam também ter
    retornado bool, em vez de int */

   int operator==(char *s) {return !strcmp(p, s);}
   int operator!=(char *s) {return strcmp(p, s);}
   int operator<(char *s) {return strcmp(p, s) < 0;}
   int operator>(char *s) {return strcmp(p, s) > 0;}
   int operator<=(char *s) {return strcmp(p, s) <= 0;}
```

```
  int operator>=(char *s) {return strcmp(p, s) >= 0;}

  int strsize(void) {return strlen(p);}  // retorna o tamanho da string
  void makestr(char *s) {strcpy(s, p);} // cria string delimitada por
   aspas a partir do objeto Strings

  operator char *(void) {return p;} // conversão para char
}
```

A maioria das funções-membro dentro da classe *Strings* é relativamente auto-explicativa porque está construída com base nas técnicas de sobrecarga de operadores que você aprendeu em dicas anteriores. No entanto, observe que a classe *Strings* contém somente dois membros privados, *p* e *tamanho*. Conforme você aprenderá na dica a seguir, quando criar um objeto *Strings*, a função construtora usará *new* para alocar memória dinamicamente e colocar o ponteiro para a memória no membro *p*. O membro *tamanho* conterá um inteiro que representa o comprimento da string.

ESCREVENDO AS CONSTRUTORAS PARA A CLASSE STRINGS 1166

Como você aprendeu na declaração da classe *Strings* que criou na dica anterior, a classe *Strings* suporta três construtoras diferentes: uma construtora não-inicializada, uma construtora que espera uma string delimitada por aspas como sua inicializadora, e uma construtora que espera outro objeto *Strings* como sua inicializadora. Como você verá, a construtora efetuará essencialmente o mesmo processamento em ambos os últimos dois casos. No entanto, você precisa criar construtoras que permitam ao seu programa fornecer todas as possibilidades (exatamente como você viu com os operadores sobrecarregados na dica anterior). O CD-ROM que acompanha este livro inclui o arquivo *strings.cpp*, que inclui a declaração da dica anterior nesta dica, como mostrado aqui:

```
Strings::Strings(void)
 {
   tamanho = 1;
   p = new char[tamanho];
   if(!p)
    {
      cout << "Erro de alocação!" << endl;
      exit(1);
    }
   *p = '\0';
 }

Strings::Strings(char *str)
 {
   tamanho = strlen(str) + 1;
   p = new char[tamanho];
   if(!p)
    {
      cout << "Erro de alocação!" << endl;
      exit(1);
    }
   strcpy(p, str);
 }

Strings::Strings(const Strings &obj)
 {
   tamanho = obj.tamanho;
   p = new char[tamanho];
   if(!p)
    {
      cout << "Erro de alocação!" << endl;
      exit(1);
    }
   strcpy(p, obj.p);
```

```
}
```

Como você pode ver, cada construtora aloca memória, cria uma matriz *char* do tamanho solicitado e inicializa *p* para apontar para o início da matriz. Quando o programa destrói o objeto *Strings*, a função destrutora in-line simplesmente libera a memória para a qual *p* aponta.

1167 EFETUANDO E/S COM A CLASSE STRINGS

Na Dica 1165, você criou a definição básica para a classe *Strings*. A definição incluía referências para duas funções *friend* que sobrecarregam os operadores de inserção e extração, como mostrado aqui:

```
friend ostream &operator<<(ostream, &canal, Strings &obj);
friend istream &operator>>(istream, &canal, Strings &obj);
```

Como entrada e saída serão as operações mais comuns que seus programas efetuarão com as strings, você precisará implementar uma versão sobrecarregada dentro da classe. O canal de saída trata o objeto *Strings* facilmente, enviando as informações diretamente para o canal. Observe que o extrator recebe o objeto *Strings* por referência. Como os objetos *Strings* podem ser conceitualmente muito grandes, melhorará o desempenho do seu programa passar o objeto *Strings* como um ponteiro, em vez de por valor, como segue:

```
ostream &operator<<(ostream &canal, Strings &obj)
 {
   canal << obj.p;
   return canal;
 }
```

1168 ESCREVENDO AS FUNÇÕES DE ATRIBUIÇÃO PARA A CLASSE STRINGS

Como você viu na Dica 1165, escrever as funções de atribuição para a classe *Strings* é uma tarefa repetitiva, similar a escrever as funções construtoras *Strings*. Os dois operadores sobrecarregados precisam atribuir o objeto *Strings* ou o valor de outro objeto *Strings* ou o valor de uma string delimitada por aspas e precisam tratar as duas situações de forma ligeiramente diferente. Como você pode ver, em ambos os casos, o operador de atribuição sobrecarregado limpa a memória atualmente alocada para *p*, cria nova memória para *p*, e, depois, atribui o novo valor para a nova memória. A única diferença significativa entre as duas funções é que a primeira recebe um operando do tipo *Strings* e a segunda recebe um operando do tipo *char* *, como mostrado aqui:

```
Strings Strings::operator=(Strings &obj)
  {
   Strings temp(obj.p);

   if(obj.tamanho > tamanho)
    {
      delete p;
      p = new char[obj.tamanho];
      tamanho = obj.tamanho;
      if(!p)
       {
         cout << "Erro de alocação!" << endl;
         exit(1);
       }
    }
   strcpy(p, obj.p);
   strcpy(temp.p, obj.p);
   return temp;
 }

Strings Strings::operator=(char *s)
  {
```

```
  int tam = strlen(s) + 1;

  if(tamanho < tam)
   {
     delete p;
     p = new char[tam];
     tamanho = tam;
     if(!p)
      {
        cout << "Erro de alocação!" << endl;
        exit(1);
      }
   }
  strcpy(p, s);
  return *this;
 }
```

Ambas as versões sobrecarregadas do operador = usam a função *strcpy* para colocar o valor da string *rvalue* em *lvalue*. Como a atribuição modifica diretamente a string *lvalue*, ambas as funções usam o ponteiro *this* para atribuir um novo valor ao objeto *Strings* da esquerda.

SOBRECARREGANDO O OPERADOR + PARA CONCATENAR OBJETOS STRINGS 1169

Como você determinou na Dica 1164, sua classe *Strings* deve lhe permitir usar o operador + para concatenar dois objetos *Strings*. Exatamente como você sobrecarregou o operador de atribuição para permitir que a classe *Strings* tratasse ambos os outros objetos *Strings* e strings de caracteres delimitadas por aspas, você também precisa sobrecarregar o operador + para tratar ambas as situações. No entanto, como o operador + é um operador binário e não um operador unário, você precisa criar três versões sobrecarregadas para tratar todas as atividades de concatenação: uma que trate dois objetos *Strings*, uma que trate um objeto *Strings* e uma string delimitada por aspas a sua direita, e uma que trate uma string delimitada e um objeto *Strings* a sua direita. Como você aprendeu, pode usar uma função *friend* de gabarito para evitar reescrever o mesmo algoritmo três vezes, mas você não fará isso aqui porque é útil compreender cada situação que seu código precisa tratar. Como com as dicas anteriores, o CD-ROM que acompanha este livro contém dentro do arquivo *strings.cpp* o código para a implementação das funções do operador + sobrecarregadas. Você implementará essas funções como mostrado aqui:

```
Strings Strings::operator+(Strings &obj)
 {
   int tam;
   Strings temp;

   delete temp.p;
   tam = strlen(obj.p) + strlen(p) + 1;
   temp.p = new char[tam];
   temp.tamanho = tam;
   if(!temp.p)
    {
      cout << "Erro de alocação!" << endl;
      exit(1);
    }
   strcpy(temp.p, p);
   strcat(temp.p, obj.p);
   return temp;
 }

Strings Strings::operator+(char *s)
 {
   int tam;
   Strings temp;
```

```
    delete temp.p;
    tam = strlen(s) + strlen(p) + 1;
    temp.p = new char[tam];
    temp.tamanho = tam;
    if(!temp.p)
     {
       cout << "Erro de alocação!" << endl;
       exit(1);
     }
    strcpy(temp.p, this.p);
    strcat(temp.p, s);
    return temp;
 }

Strings operator+(char *s, Strings &obj)
 {
    int tam;
    Strings temp;

    delete temp.p;
    tam = strlen(s) + strlen(obj.p) + 1;
    temp.p = new char[tam];
    temp.tamanho = tam;
    if(!temp.p)
     {
       cout << "Erro de alocação!" << endl;
       exit(1);
     }
    strcpy(temp.p, s);
    strcat(temp.p, obj.p);
    return temp;
 }
```

Observe que a terceira função sobrecarregada é uma função *friend* em vez de um operador-membro sobrecarregado. Como você aprendeu, quando usa um operador-membro, ele explicitamente passa o objeto na esquerda — o que causará um erro se o objeto na esquerda não for do tipo de classe correto. Quando você adiciona o objeto *Strings* a uma string delimitada por aspas, precisa, em vez disso, passar a string explicitamente e anexar o objeto ao valor da string delimitada por aspas.

1170 REMOVENDO UMA STRING A PARTIR DE DENTRO DE UM OBJETO STRINGS

Uma função string útil que você irá querer incluir em seu objeto *Strings* é a subtração de substrings. Quando o objeto *Strings* implementa a operação de subtração de substring, ele remove todas as ocorrências da substring do objeto *Strings*. Para compreender melhor como *Strings* implementa a operação de subtração de substring, considere o seguinte fragmento de código:

```
amostra = "Bíblia do Programador C/C++";
amostra = amostra - "C";
// amostra agora é igual a "Bíblia do Programador /++"
```

Ao contrário das três funções sobrecarregadas que você implementou para sobrecarregar o operador +, você usará somente duas funções sobrecarregadas para sobrecarregar o operador – (pois você não subtrairá um objeto *Strings* de uma string delimitada por aspas). O CD-ROM que acompanha este livro inclui o código de implementação dentro do arquivo *string.cpp*, como segue:

```
Strings Strings::operator-(Strings &substr)
 {
```

```
    Strings temp(p);
    char *s1;
    int i,j;

    s1 = p;
    for(i=0; *s1; i++)
     {
       if(*s1!=*substr.p)
        {
          temp.p[i] = *s1;
          s1++;
        }
       else
        {
          for(j=0; substr.p[j]==s1[j] && substr.p[j]; j++)
             ;
          if(!substr.p[j])
           {
             s1 += j;
             i--;
           }
          else
           {
             temp.p[i] = *s1;
             s1++;
           }
        }
     }
    temp.p[i] = '\0';
    return temp;
 }

Strings Strings::operator-(char *substr)
 {
    Strings temp(p);
    char *s1;
    int i,j;

    s1 = p;
    for(i=0; *s1; i++)
     {
       if(*s1!=*substr)
        {
          temp.p[i] = *s1;
          s1++;
        }
       else
        {
          for(j=0; substr[j]==s1[j] && substr[j]; j++)
             ;
          if(!substr[j])
           {
             s1 += j;
             i--;
           }
          else
           {
             temp.p[i] = *s1;
             s1++;
           }
        }
```

```
    }
    temp.p[i] = '\0';
    return temp;
}
```

Ambas as funções sobrecarregadas copiam o conteúdo do operando esquerdo na variável temp. À medida que cada função copia o operando esquerdo, ela remove quaisquer ocorrências da substring que o operando direito especifica durante o processo. As funções de operadores sobrecarregados então retornam o objeto *Strings* resultante. Como você definiu a classe *Strings* até aqui, todas as instruções a seguir são válidas para uso com o operador de subtração *Strings*.

```
Strings x("ABCABCD"), y("A");
Strings z;
z = x - y;      // z = "BCBCD"
```

1171 SOBRECARREGANDO OS OPERADORES RELACIONAIS

Em dicas anteriores você sobrecarregou muitos operadores que seus programas poderiam usar com eficácia para manipular strings. Outro aspecto importante do poder da classe *Strings* é sua capacidade de comparar duas strings facilmente e retornar um resultado mais significativo que -1, 0 ou 1 (os valores de retorno da função *strcmp*). No entanto, os operadores relacionais para a classe *Strings*, que retornam tais valores, são simples. Eles usam a função *strcmp* com o membro *p* no lado esquerdo do operador e um objeto ou um ponteiro *char* para avaliar o lado direito do operador. Devido à simplicidade das atividades que os operadores relacionais executam, a definição de classe *Strings* define todos os operadores relacionais in-line dentro da definição da classe. As definições do operador relacional estão dentro da classe *strings.cpp* e são mostradas aqui:

```
// operadores relacionais entre objetos Strings. Observe
// que os operadores poderiam também retornar bool, em
// vez de int
   int operator==(Strings &obj) {return !strcmp(p, obj.p);}
   int operator!=(Strings &obj) {return strcmp(p, obj.p);}
   int operator<(Strings &obj) {return strcmp(p, obj.p) < 0;}
   int operator>(Strings &obj) {return strcmp(p, obj.p) > 0;}
   int operator<=(Strings &obj) {return strcmp(p, obj.p) <= 0;}
   int operator>=(Strings &obj) {return strcmp(p, obj.p) >= 0;}

 // operadores relacionais entre objeto Strings e uma
 // string  de caracteres delimitada por aspas. Observe
 // que os operadores poderiam também retornar bool, em
 // vez de int

   int operator==(char *s) {return !strcmp(p, s);}
   int operator!=(char *s) {return strcmp(p, s);}
   int operator<(char *s) {return strcmp(p, s) < 0;}
   int operator>(char *s) {return strcmp(p, s) > 0;}
   int operator<=(char *s) {return strcmp(p, s) <= 0;}
   int operator>=(char *s) {return strcmp(p, s) >= 0;}
```

Todas as definições para os operadores relacionais dentro da classe *Strings* assumem que você irá comparar um objeto *Strings* com outro objeto *Strings* ou uma string delimitada por aspas. Se quiser que sua classe *Strings* permita que seus programas comparem strings delimitadas por aspas com um objeto *Strings*, você precisará definir outro conjunto de funções amigas para sobrecarregar os operadores relacionais novamente. Por exemplo, você declararia a sobrecarga do terceiro operador == como segue:

```
friend Strings operator==(char *s, Strings &obj);
```

Fora da definição da classe, você precisa criar uma função de operador sobrecarregado, como mostrado aqui:

```
Strings operator==(char * s, Strings &obj)
 {
   return !strcmp(s, obj.p);
 }
```

DETERMINANDO O TAMANHO DE UM OBJETO STRINGS 1172

Uma das atividades mais comuns que você executará com strings é determinar seu tamanho atual. Normalmente, você usará o tamanho atual de uma string para efetuar processamento adicional na string, para formatar a saída, e assim por diante. Para lhe ajudar a determinar o tamanho de um objeto *Strings*, a classe *Strings* fornece a seguinte função-membro:

```
int strsize(void)    {return strlen(p);} // retorna o tamanho da string
```

Quando você chama a função-membro *strsize* em um objeto *Strings*, a função-membro retorna um valor inteiro que representa o comprimento do objeto *Strings*. Você chamará a função-membro *Strings* em um objeto *Strings*, a seguir:

```
Strings amostra;
int x;
x = amostra.strsize();
```

CONVERTENDO UM OBJETO STRINGS PARA UMA MATRIZ DE CARACTERES 1173

A definição da classe *Strings* fornece duas funções-membro que seus programas podem usar para manipular a matriz *char* dentro de um objeto *Strings* sem manipular o próprio objeto. A primeira função, *makestr*, copia a matriz *char* dentro de um objeto *Strings* para uma matriz *char* padrão. Mais freqüentemente você usará a função *makestr* para obter uma string terminada por *NULL* a partir de um objeto *Strings*. Seus programas usarão a função-membro *makestr* em um objeto *Strings*, como mostrado aqui:

```
Strings amostra = "Bíblia do Programador C/C++"
char matriz[256];

amostra.makestr(matriz);
```

Com a função-membro *makestr*, você pode escolher entre matrizes *char* e *Strings*, e facilmente converter de uma para a outra.

USANDO UM OBJETO STRINGS COMO UMA MATRIZ DE CARACTERES 1174

Na dica anterior, você aprendeu que a classe *Strings* suporta duas funções-membro que lhe ajudam a usar seus objetos *Strings* como matrizes *char* dentro de seu programa. A primeira função-membro, *makestr*, copia o componente string do objeto *Strings* para uma matriz *char*. A segunda função-membro, uma sobrecarga do operador (), lhe permite usar diretamente um objeto *Strings* dentro de qualquer função que espera uma matriz de caracteres normal terminada por *NULL*. Por exemplo, o código a seguir compilará e funcionará corretamente:

```
Strings x("olá");
puts(x);
```

Como você quer evitar sobrecarregar toda função que usa ou espera usar uma matriz ou ponteiro *char*, sobrecarregar o operador () para que ele retorne uma matriz *char*, em vez de retornar informações específicas sobre o objeto *Strings*, mantém seu código mais claro e torna seu objeto *Strings* mais útil.

1175 DEMONSTRANDO O OBJETO STRINGS

Usar a classe *Strings* que você criou nas dicas anteriores é surpreendentemente fácil. Após você completar a definição, usar as muitas capacidades dos objetos *Strings* é um processo simples. O fragmento de código a seguir usa sua definição de classe *Strings* e demonstra algumas das muitas capacidades da classe:

```
void main(void)
  {
   Strings s1("Um programa de exemplo que usa objetos string.\n");
   Strings s2(s1);
   Strings s3;
   char s[80];

   cout << s1 << s2;
   s3 = s1;
   cout << s3;
   s3.makestr(s);
   cout << "Convertida para uma string: " << s;

   s2 = "Isto é uma nova string.";
   cout << s2 << endl;

   Strings s4("Isto também é uma nova string.");
   s1 = s2 + s4;
   cout << s1 << endl;

   if(s2 == s3)
      cout << "As strings são iguais." << endl;
   if(s2 != s3)
      cout << "As strings não são iguais." << endl;
   if(s1 < s4)
      cout << "s1 é menor que s4." << endl;
   if(s1 > s4)
      cout << "s1 é maior que s4." << endl;
   if(s1 <= s4)
      cout << "s1 é menor ou igual a s4." << endl;
   if(s1 > s4)
      cout << "s1 é maior ou igual a s4." << endl;

   if(s2 > "ABC")
      cout << "s2 é maior que 'ABC'" << endl << endl;

   s1 = "um dois três um dois três\n";
   s2 = "dois";
   cout << "String inicial: " << s1;
   cout << "String após a subtração de dois: ";
   s3 = s1 - s2;
   cout << s3;

   cout << endl;
   s4 = "Bíblia do Programador ";
   s3 = s4 + "C/C++, do Jamsa!\n";
   cout << s3;
   s3 = s3 - "C/C++";
   s3 = "Esta é " + s3;
   cout << s3;

   cout << "Digite uma string: ";
   cin  >> s1;
   cout << s1 << endl;
   cout << "s1 tem " << s1.strsize() << " caracteres de comprimento." << endl;
```

```
    puts(s1);

    s1 = s2 = s3;
    cout << s1 << s2 << s3;
    s1 = s2 = s3 = "Programa finalizado.\n";
    cout << s1 << s2 << s3;
}
```

A função *main* impressa dentro do fragmento de código anterior aparece dentro do programa *usa_str.cpp* no CD-ROM que acompanha este livro.

CRIANDO UM CABEÇALHO PARA A CLASSE STRINGS 1176

Em dicas anteriores você criou uma útil e extensa classe *Strings*. Caso seu compilador não inclua uma definição de classe *string*, ou se você achar que gosta mais da sua classe *Strings*, você pode querer mover a definição da classe para dentro de um arquivo de cabeçalho para que ela não ocupe muito espaço dentro de seus programas. Para converter a classe *Strings* em um arquivo de cabeçalho, exclua *main* e todos os seus componentes de *usa_str.cpp*. Em seguida, salve o código remanescente em um arquivo de cabeçalho chamado *strings.h*. Salve o arquivo de cabeçalho dentro do diretório *include* do seu compilador. Posteriormente, quando você quiser usar a classe Strings dentro de seus programas, poderá simplesmente referenciar o arquivo de cabeçalho do interior do código do seu programa, como mostrado aqui:

```
#include <strings.h>
```

OUTRO EXEMPLO DE STRINGS 1177

Na Dica 1175, você escreveu um programa que implementou sua classe *Strings*. Na Dica 1176, você aprendeu como criar um arquivo de cabeçalho para sua classe *Strings*. Para lhe ajudar a compreender o poder do suporte de C++ para a criação personalizada de arquivos de cabeçalhos, vale a pena escrever outro programa que usa o novo arquivo de cabeçalho *strings.h*. O programa desta dica, *chec_arq cpp*, aceita um único parâmetro da linha de comando, que corresponde ao nome de um programa executável que você quer localizar. O programa tentará abrir um arquivo de programa na unidade de disco local que tem o mesmo nome. Se o programa for bem-sucedido, ele informará que o arquivo existe na unidade de disco local. Se não for bem-sucedido, o programa informará que o arquivo não existe. Você implementará o programa *chec_arq.cpp* como a seguir:

```
#include "strings.h"
#include <iostream.h>
#include <fstream.h>

// extensões de arquivo executável

char ext[3][4] = { "EXE", "COM", "BAT" };

void main(int argc, char *argv[])
 {
    Strings nomearq;
    int i;

    if(argc != 2)
    {
        cout << "Uso nomearq nome" << endl;
        exit(1);
    }
    nomearq = argv[1];
    nomearq = nomearq + ".";
    for (i = 0; i < 3; i++)
    {
        nomearq = nomearq + ext[i];
        cout << "Testando " << nomearq << " ";
```

```
        ifstream f; f.open(nomearq);
        if(f)
        {
            cout << " - Existe" << endl;
            f.close();
        }
        else
            cout << " - Não-encontrado"<< endl;
        nomearq = nomearq - ext[i];
    }
}
```

É especialmente importante observar que o compilador aceita a chamada *f.open* mesmo que receba um objeto *Strings*, não um ponteiro *char*. O compilador aceita a chamada *f.open* porque sua sobrecarga do operador () dentro da classe *Strings* permite que o compilador interprete as informações como *char**, em vez de um objeto *Strings*. Quando você compilar e executar o programa *chec_arq.cpp*, sua tela exibirá a saída similar à seguinte:

```
C:\> chec_arq teste
teste.EXE  -  Não-encontrado
teste.OBJ  -  Não-encontrado
teste.COM  -  Não-encontrado

C:\> chec_arq chec_arq
chec_arq.EXE - Encontrado
chec_arq.OBJ - Encontrado
chec_arq.COM - Encontrado
C:\>
```

1178 USANDO UMA CLASSE DE C++ PARA CRIAR UMA LISTA DUPLAMENTE LIGADA

Em dicas recentes você expandiu seu conhecimento de como deve criar e usar suas próprias classes. Na série de exemplos *Strings*, você aprendeu como derivar sua própria classe e usá-la em seus programas. Outro aspecto útil das classes é sua habilidade em tratar vários papéis diferentes para lhe ajudar a processar seus dados. Por exemplo, na Dica 749 você criou a estrutura *ListaEnt* para lhe ajudar a processar uma lista duplamente ligada, como mostrado aqui:

```
struct ListaEnt {
   int numero;
   struct ListaEnt *proximo;
   struct ListaEnt *anterior;
  } inicio, *nodo;
```

Entretanto, para efetuar operações em qualquer nodo dentro da lista (tal como percorrer a lista), você precisava acessar uma série de funções exteriores. Nas dicas a seguir, você aprenderá como construir a classe *lista_objeto*, que poderá usar dentro de seus programas para manter informações sobre as listas duplamente ligadas.

1179 COMPREENDENDO OS MEMBROS DA CLASSE DBLLINKOB

Na dica anterior, você revisitou rapidamente a estrutura *ListaEnt* que já havia criado. Quando você trabalhar com a classe *lista_objeto*, os membros serão ligeiramente diferentes. Na verdade, você não acrescentará qualquer novo membro de dados; os membros restantes que você acrescentará à classe *lista_objetos* serão os cabeçalhos para uma série de funções que seus programas usarão para navegar e manipular melhor a lista ligada. O CD-ROM que acompanha este livro inclui os membros da classe *lista_objeto* dentro do arquivo *dblinkcl.cpp*, como segue:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
```

```
class lista_objeto {
 public:
   char info;
   lista_objeto *proximo;
   lista_objeto *anterior;
   lista_objeto(void)  {
      info = 0;
      proximo = NULL;
      anterior = NULL;
    }
   lista_objeto *pegaproximo(void)   {return proximo;}
   lista_objeto *pegaanterior(void)  {return anterior;}
   void pegainfo(char &c)  { c = info; }
   void muda(char c)   {info = c;}
   friend ostream &operator << (ostream &canal, lista_objeto o)
    {
      canal << o.info << endl;
      return canal;
    }
   friend ostream &operator << (ostream &canal, lista_objeto *o)
    {
      canal << o->info << endl;
      return canal;
    }
   friend istream &operator >> (istream &canal, lista_objeto &o)
    {
      cout << "Digite as informações: " << endl;
      canal >> o.info;
      return canal;
    }
};
```

Uma coisa importante a observar com relação à definição *lista_objeto* é como ela define suas funções *amigas* (que ela usa para manipular a entrada e a saída nos canais) in-line dentro da definição da classe, uma construção ligeiramente diferente daquelas que você usou antes. Você poderia também declarar as funções *amigas* como faz normalmente. Por exemplo, você poderia declarar o operador de inserção sobrecarregado, como mostrado aqui:

```
friend istream &operator>>(istream &canal, lista_objeto &o);

// Código da classe

istream &operator>>(istream &canal, lista_objeto &o);
  {
    cout << "Digite as informações: " << endl;
    stream >> o.info;
    return canal;
  }
```

COMPREENDENDO AS FUNÇÕES PEGAPROXIMO E PEGAANTERIOR 1180

Na definição da classe que você viu na dica anterior, existiam duas definições de funçao in-line para as funções *pegaproximo* e *pegaanterior*. Exatamente como, com a lista duplamente ligada que você criou anteriormente usando a estrutura *ListaEnt*, cada ocorrência da classe *lista_objeto* contém dois ponteiros: um para o item antes do objeto na lista e uma para o item que segue o objeto na lista. Em vez de chamar funções de fora, como você fez com a lista com base em estruturas, a lista com base em classe lhe permite criar as funções de movimento dentro da própria classe, como mostrado aqui:

```
lista_objeto *pegaproximo(void)   {return proximo;}
lista_objeto *pegaanterior(void)  {return anterior;}
```

Portanto, quando seus programas navegarem na lista, eles usarão os métodos *pegaproximo* e *pegaanterior* para mover de um nodo na lista para o próximo, como segue:

```
nodo = nodo.pegaproximo();
nodo = nodo.pegaproximo();
```

1181 COMPREENDENDO AS FUNÇÕES DE SOBRECARGA OPERATOR

Ao contrário da classe *Strings*, que requeria que você sobrecarregasse a maior parte dos operadores relacionais e computacionais, a lista duplamente ligada requer somente que você sobrecarregue os operadores de entrada e de saída. No caso particular da lista ligada que você está construindo, os operadores *stream* precisam poder tratar três situações: uma *lista_objeto*, um ponteiro para uma *lista_objeto* e entrada para uma *lista_objeto*. A primeira e a última situações são relativamente intuitivas e não requerem explicações além de que ambas manipulam o membro de dados *info* (um membro *char* nesta situação). No entanto, você pode se perguntar por que a definição de classe requer a segunda função extratora sobrecarregada. Como deve se lembrar do seu trabalho anterior com uma lista ligada, você frequentemente manipulará os ponteiros para os outros objetos na lista — assim, criar uma função extratora capaz de tratar um ponteiro para um objeto *lista_objeto* é útil. Você implementará as três funções de sobrecarga de operador dentro de seus programas como mostrado aqui:

```
friend ostream &operator<<(ostream &canal, lista_objeto o)
 {
   canal << o.info << endl;
   return canal;
 }
friend ostream &operator<<(ostream &canal, lista_objeto *o)
 {
   canal << o->info << endl;
   return canal;
 }
friend istream &operator>>(istream &canal, lista_objeto &o)
 {
   cout << "Digite as informações: " << endl;
   canal >> o.info;
   return canal;
 }
```

As duas primeiras funções sobrecarregadas exibem na tela aquilo que o programa armazenou anteriormente dentro do membro *info* de nodo. A terceira função lhe permite inserir informações no membro *info* de nodo. Você deve prestar atenção em como a definição *lista_objeto* sobrecarrega os operadores de canal — *lista_objeto* define suas funções *amigas* (que ela usa para manipular a entrada e a saída nos canais) in-line dentro da definição da classe. A definição *lista_objeto* usa uma construção ligeiramente diferente da que você usou antes, mas você poderia, da mesma forma, declarar as funções *amigas* como normalmente faz. Por exemplo, poderia declarar o operador de inserção sobrecarregado, como mostrado aqui:

```
friend istream &operator>>(istream &canal, lista_objeto &o);

   // Código da classe
   istream &operator>>(istream &canal, lista_objeto &o);
   {
      cout << "Digite as informações: " << endl;
      canal >> o.info;
      return canal;
   }
};
```

HERDANDO A CLASSE LISTA_OBJETO 1182

Em dicas anteriores você definiu e estudou a classe *lista_objeto* a partir da qual criará objetos dentro de uma lista duplamente ligada. No entanto, você precisa compreender que a classe *lista_objeto* somente define informações sobre cada objeto dentro da lista — a classe *lista_objeto* não fornece a seus programas informações sobre a lista propriamente. Na próxima dica você irá derivar a classe *lista_ligada* a partir da classe *lista_objeto* para manter informações sobre a própria lista. No entanto, antes de derivar a classe *lista_ligada*, certifique-se de que compreende o relacionamento entre cada objeto e a lista, como ilustra a Figura 1182.

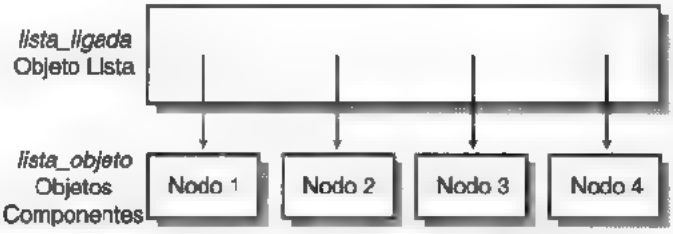

Figura 1182 O relacionamento entre as classes *lista_objeto* e *lista_ligada*.

COMPREENDENDO A CLASSE LISTA_LIGADA 1183

Como você aprendeu na dica anterior, a classe *lista_objeto* não fornece ao programa informações sobre a lista, somente sobre cada item na lista. Para manter as informações sobre uma dada lista de itens, você precisa derivar uma segunda classe, a classe *lista_ligada*, a partir da classe *lista_objeto*. A classe *lista_ligada* precisa manter dois ponteiros, um para o início da lista e um para o item final da lista. *Inicio* e *fim* são ambos os ponteiros para objetos *lista_objeto*. A construtora *lista_ligada* inicializa ambos os ponteiros com *NULL* cada vez que seu programa cria uma nova lista. Você implementará a classe *lista_ligada* inteira, como mostrado aqui:

```
class lista_ligada : public lista_objeto  {
    lista_objeto *inicio, *fim;
  public:
    lista_ligada(void)   {inicio = fim = NULL;}
    void armazena(char c);
    void remove(lista_objeto *obj);
    void frwdlist(void);
    void bkwdlist(void);
    lista_objeto *localiza(char c);
    lista_objeto *pegainicio(void)   {return inicio;}
    lista_objeto *pegafim(void)      {return fim;}
};
```

Além dos ponteiros *inicio* e *fim*, a classe *lista_ligada* implementa várias funções que seus programas podem usar para negociar e manipular a lista. As funções-membro adicionais permitem que seus programas executem as seguintes ações:

- Colocar um item na lista
- Remover um item da lista
- Exibir a lista na ordem do começo para o fim e do fim para o começo
- Localizar um elemento específico na lista
- Obter ponteiros para o início e fim da lista

As dicas a seguir examinam a implementação de cada ação na lista anterior em detalhes.

1184 COMPREENDENDO A FUNÇÃO ARMAZENA DA LISTA_LIGADA

Como você aprendeu, é possível usar classes para implementar de forma mais fácil uma lista duplamente ligada similar àquela que você criou em dicas anteriores. Como sabe, uma das mais importantes ações que seus programas precisam efetuar com qualquer lista ligada é a inserção de objetos. Na classe *lista_ligada*, a função-membro *armazena* trata a inserção de objetos na lista. Você implementará a função-membro *armazena* dentro de sua classe *lista_ligada* da seguinte forma:

```
void lista_ligada::armazena(char c)
 {
   lista_objeto *p;

   p = new lista_objeto;
   if(!p)
    {
       cout << "Erro de alocação." << endl;
       exit(1);
    }
   p->info = c;
   if(inicio == NULL)
   {
      fim = inicio = p;
   }
   else
   {
      p->anterior = fim;
      fim->proximo = p;
      fim = p;
   }
 }
```

Antes de seu programa poder inserir um novo item (neste caso, um *char*) na lista, a função precisa criar uma nova *lista_objeto* para armazenar o item. A função *armazena* tentará criar um novo objeto *lista_objeto* e encerrará o programa se não conseguir fazer isso. Se for bem-sucedida, a função *armazena* armazenará o valor em seu parâmetro *c* dentro do objeto *lista_objeto* recém-criado. A função *armazena* adicionará então a nova *lista_objeto* ao final da lista. A função também atualizará os ponteiros *inicio* e *fim* do objeto *lista_ligada*, como é apropriado. Conforme implementada dentro de sua classe *lista_ligada*, *armazena* sempre adicionará novos membros ao final da lista. No entanto, você pode modificar facilmente a função ou a classe para que *armazena* insira novos objetos na posição correta dentro da lista para criar uma lista classificada, como você fez com outras listas ligadas em dicas anteriores.

Como a função *armazena* mostra claramente, *lista_ligada* gerencia um conjunto de objetos do tipo *lista_objeto*. O tipo de dados que a lista armazena é irrelevante para a classe *lista_ligada*. Em outras palavras, você precisa modificar somente a classe *lista_objeto* para suportar o armazenamento de dados mais úteis. Como você aprenderá na Dica 1191, a classe *lista_ligada* pode ser utilizada em uma construção genérica, que você usará para criar múltiplas listas ligadas de múltiplos tipos em um único programa.

1185 COMPREENDENDO A FUNÇÃO REMOVE DE LISTA_LIGADA

Na dica anterior você acrescentou a função *armazena* na classe *lista_ligada*. A função *armazena* lhe permite acrescentar novos objetos *lista_objeto* na sua lista. Outra tarefa importante que seu gerenciador *lista_ligada* precisa executar é remover objetos *lista_objetos* de dentro da lista. A classe *lista_ligada* efetua a remoção de objetos dentro da função-membro *remove*, que você implementará da seguinte forma:

```
void lista_ligada::remove(lista_objeto *ob)
 {
   if(ob->anterior)
    {
```

```
      ob->anterior->proximo = ob->proximo;
      if(ob->proximo)
         ob->proximo->anterior = ob->anterior;
      else
         fim = ob->anterior;
   }
  else
   {
     if(ob->proximo)
      {
        ob->proximo->anterior = NULL;
        inicio = ob->proximo;
      }
     else
        inicio = fim = NULL;
   }
}
```

A função *remove* exclui o objeto apontado pelo ponteiro ob dentro da lista. Como você aprendeu no exemplo da lista anterior, um objeto que você precisa excluir da lista reside em um dos três lugares dentro da lista: ele é o primeiro item, o último item ou um item no meio, como mostrado na Figura 1185.

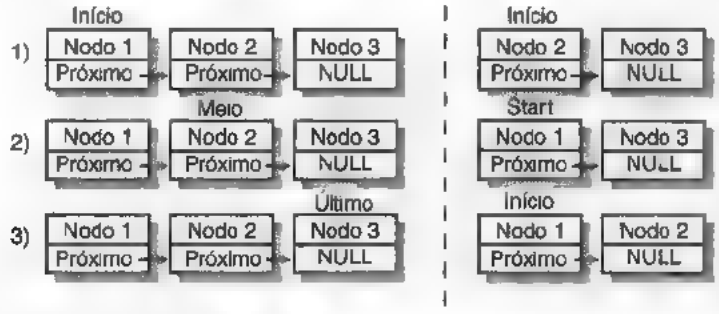

Figura 1185 Os três possíveis casos que *remove* trata.

A função-membro *remove* trata todas as possibilidades e condensa a lista após remover o objeto que você não deseja mais

COMPREENDENDO AS FUNÇÕES PEGAINICIO E PEGAFIM 1186

As duas únicas funções in-line que a classe *lista_ligada* define são *pegainicio* e *pegafim*, mostradas aqui:

```
lista_objeto *pegainicio(void)  {return inicio;}
lista_objeto *pegafim(void)     {return fim;}
```

Tanto *pegainicio* e *pegafim* são puramente funções de interface. Você pode chamar *pegainicio* e *pegafim* a partir de qualquer lugar no programa para mover o ponteiro da lista para o início ou para o fim da lista. Outras funções-membro dentro da classe também podem chamar *pegainicio* e *pegafim* para ajudar as funções a percorrer a lista, como você aprenderá na próxima dica. Ambas as funções *frentelista* e *traslista* usam *pegainicio* e *pegafim* para se posicionar dentro da lista antes que *frentelista* e *traslista* gerem saída.

EXIBINDO A LISTA_LIGADA NA ORDEM DO COMEÇO PARA O FIM 1187

Uma atividade comum que seus programas executarão com qualquer lista ligada é exibir a lista. Na dica a seguir, você aprenderá como exibir uma lista de trás para a frente. No entanto, seus programas freqüentemente exibirão uma lista na ordem em que o programa a criou. Para suportar a exibição de uma lista do começo para o final, a classe *lista_ligada* define a função *frentelista*, como segue:

```
void lista_ligada::frentelista(void)
 {
   lista_objeto *temp;

   temp = pegainicio;
   do {
      cout << temp->info << " ";
      temp = temp->pegaproximo();
    } while(temp);
   cout << endl;
 }
```

A função *frentelista* não aceita parâmetros porque sempre percorre a lista inteira. Quando você invocar *frentelista* para um objeto *lista_ligada*, *frentelista* primeiro usará a função *pegainicio* para obter o ponteiro para o primeiro objeto na lista. Em seguida, a função *frentelista* usará um laço *do* e a função *pegaproximo* para percorrer a lista, um item de cada vez, e exibirá os dados que cada item contém, em seqüência.

1188 EXIBINDO A LISTA_LIGADA EM ORDEM INVERSA

Na dica anterior, você criou a função-membro *frentelista*, que seus programas podem usar para exibir uma *lista_ligada* do primeiro para o último. Como você sabe, seus programas precisam com frequência exibir uma lista duplamente ligada em ordem inversa. Para ajudar seus programas a exibir listas em ordem inversa, a classe *lista_ligada* fornece a função *traslista*, como mostrado aqui:

```
void lista_ligada::traslista(void)
 {
   lista_objeto *temp;
   temp = pegafim;
   do {
      cout << temp->info << " ";
      temp = temp->pegaanterior();
   } while(temp);
   cout << endl;
 }
```

Exatamente como a função *frentelista* usa a função *pegainicio* para obter um ponteiro para o primeiro membro na lista e depois percorre a lista do início para o fim, a função *traslista* usa a função *pegafim* para obter um ponteiro para o último membro na lista. Em seguida, a função *traslista* usa um laço *do* e a função *pegaanterior* (que retorna um ponteiro para o item anterior na lista) para percorrer a lista de trás para a frente.

1189 PESQUISANDO A LISTA

Como você aprendeu nas dicas anteriores, o objeto *lista_ligada* acrescenta novos itens no final da lista. Portanto, quando você precisa encontrar um item dentro da lista, seu programa precisa poder pesquisar o item na lista, pois de outra forma você não terá um registro permanente de onde a lista armazena o objeto. Para lhe ajudar a pesquisar um item em uma lista, a função *lista_ligada* implementa a função *localiza* para ajudar seus programas a procurar um elemento dentro da lista, como segue:

```
lista_objeto *lista_ligada::localiza(char c)
 {
   lista_objeto *temp;

   temp = pegainicio();
   while(temp)  {
      if(c == temp->info)
        return temp;
      temp = temp ->pegaproximo();
    }
   return NULL;
 }
```

A função *localiza* inicia seu processamento chamando a função *pegainicio*. Após *localiza* obter um ponteiro para o primeiro objeto na lista, ele percorre a lista um objeto de cada vez, tentando comparar o parâmetro que recebe com o item. Se *localiza* encontrar o item, retornará um ponteiro para o item; se *localiza* não encontrar o item, retornará um ponteiro *NULL*.

IMPLEMENTANDO UM PROGRAMA LISTA_LIGADA SIMPLES

Nas dicas anteriores, você criou as classes simples *lista_objeto* e *lista_ligada*. Para implementar as classes simples, seus programas precisam criar um objeto do tipo *lista_ligada* e um ponteiro para um objeto do tipo *lista_objeto*. O CD-ROM que acompanha este livro inclui o programa *usa_liga.cpp*, que implementa ambas as classes. O código a seguir mostra a função *main* do programa *usa_liga.cpp*. O programa *usa_liga.cpp* cria uma lista simples de três itens, depois percorre a lista para a frente e para trás, acrescenta e exclui itens e percorre a lista "manualmente" (usando *pegainicio* e *pegaproximo*):

```
void main(void)
 {
   dblista lista;
   char c;
   dbliga_ob *p;

   lista.armazena('1');
   lista.armazena('2');
   lista.armazena('3');

   cout << "Aqui está a lista do fim para o início, "
        << "depois do começo para o fim." << endl;
   lista.traslista();
   lista.frentelista();

   cout << endl;
   cout << "Percorrendo a lista 'manualmente'." << endl;
   p = lista.pegainicio();
   while(p) {
      p->pegainfo(c);
      cout << c << " ";
      p = p->pegaproximo();
    }
   cout << endl << endl;

   cout << "Procurando o item 2." << endl;
   p = lista.localiza('2');
   if(p)
    {
      p->pegainfo(c);
      cout << "Encontrado: " << c << endl;
    }
   cout << endl;

   p->pegainfo(c);
   cout << "Removendo o item: " << c << endl;
   lista.remove(p);
   cout << "Aqui está a nova lista do início para o fim." << endl;
   lista.frentelista();

   cout << endl;
   cout << "Acrescentando um item." << endl;
   lista.armazena('4');
   cout << "Aqui está a lista do início para o fim." << endl;
```

1190

```
    lista.frentelista();
    cout << endl;
    p = lista.localiza('1');
    if(!p)
     {
       cout << "Erro, item não-localizado." << endl;
       return 1;
     }
    p->pegainfo(c);
    cout << "Alterando " << c << " para 5." << endl;
    p->muda('5');
    cout << "Aqui está a lista do início para o fim, depois do fim
      para o começo." << endl;
    lista.frentelista();
    lista.traslista();
    cout << endl;
    cin >> *p;
    cout << p;

    cout << "Aqui está a lista do início para o fim novamente." << endl;
    lista.frentelista();
    cout << endl;
    cout << "Aqui está a lista após a remoção da cabeça da lista."
     << endl;
    p = lista.pegainicio();
    lista.remove(p);
    lista.frentelista();
    cout << endl;
    cout << "Aqui está a lista após remover fim da lista." << endl;
    p = lista.pegafim();
    lista.remove(p);
    lista.frentelista();
    return 0;
}
```

Quando você compilar e executar o programa *usa_liga.cpp*, ele exibirá a seguinte saída na sua tela:
Aqui está a lista de trás para a frente, depois da frente para trás.

```
3 2 1
1 2 3

Percorre 'manualmente' a lista.
1 2 3

Procurando o item 2.
Encontrado: 2

Removendo o item: 2
Aqui está a nova lista do início para o fim.
1 3

Acrescentando um item.
Aqui está a lista do início para o fim.
1 3 4

Alterando 1 para 5.
Aqui está a lista do início para o fim, depois do fim para o começo.
5 3 4
4 3 5

Insira as informações:
1
```

```
Aqui está a lista do início para fim novamente.
1 3 4

Aqui está a lista após remover a cabeça da lista.
3 4

Aqui está lista após remover o final da lista.
3
C:\>
```

CRIANDO UMA CLASSE DE LISTA DUPLAMENTE LIGADA GENÉRICA 1191

Em dicas anteriores você criou as classes *lista_objeto* e *lista_ligada*, que aceitaram um único *char* e o mantiveram dentro de uma lista. No entanto, como você sabe, uma classe de lista ligada é muito mais útil se aceita informações de vários tipos e armazena as informações dentro de uma lista. Por exemplo, um programa que usa três listas diferentes poderia manter uma lista com *ints*, uma lista com *floats* e uma lista que contém um tipo personalizado (tal como a classe *Livro* que você projetou em dicas anteriores). Em vez de criar uma classe *lista_objeto* e *lista_ligada* separadas para cada tipo, você deve criar uma classe *lista_objeto* e *lista_ligada* genérica. Como você aprendeu, ao criar uma classe genérica, é possível usar a classe com qualquer tipo personalizado ou C++. Sua classe de lista genérica, ou *classe gabarito*, pode tratar qualquer tipo de dado.

Uma vantagem em tornar a classe em uma classe genérica é que ela separa o *mecanismo* da lista (isto é, as funções diferentes que *lista_ligada* e *lista_objeto* implementam) dos dados que a lista realmente armazena. Usar uma classe genérica para separar o mecanismo dos dados lhe permite criar o mecanismo uma vez e usá-lo repetidas vezes.

COMPREENDENDO OS MEMBROS GENÉRICOS DA CLASSE LISTA_OBJETO 1192

Como você aprendeu na dica anterior, uma melhor implementação de suas duas classes de lista duplamente ligada é uma que usa definições genéricas. No entanto, como sua classe *lista_ligada* deriva diretamente de sua classe *lista_objeto*, você precisa primeiro tornar a classe *lista_objeto* genérica antes de poder tornar a classe *lista_ligada* genérica (como verá na dica a seguir). A implementação da classe *lista_objeto* genérica é mostrada aqui:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

template <class DataT> class lista_objeto {
 public:
   DataT info;
   lista_objeto<DataT> *proximo;
   lista_objeto<DataT> *anterior;
   lista_objeto(void)
  {
      info = 0;
      proximo = NULL;
      anterior = NULL;
    }
   lista_objeto(DataT c)
 {
      info = c;
      proximo = NULL;
      anterior = NULL;
    }
   lista_objeto<DataT> *pegaproximo(void) {return proximo;}
   lista_objeto<DataT> *pegaanterior(void)  {return anterior;}
   void pegainfo(DataT &c)  {c = info;}
```

```
    void muda(DataT c)  {info = c;}
    friend ostream &operator<<(ostream &canal, lista_objeto<DataT> o)
     {
       canal << o.info << endl;
       return canal;
     }
    friend ostream &operator<<(ostream &canal, lista_objeto<DataT> *o)
     {
       canal << o->info << endl;
       return canal;
     }
    friend istream &operator>>(istream &canal, lista_objeto<DataT> &o)
     {
       cout << "Digite as informações: " << endl;
       canal >> o.info;
       return canal;
     }
 };
```

Como você aprendeu na Dica 1120, a seguinte declaração cria a classe genérica *lista_objeto:*

```
template <class DataT> class lista_objeto {
```

A palavra-chave *template* alerta o compilador para uma descrição de classe genérica que vem em seguida. O operador *<class DataT>* permite que o compilador saiba que a classe *lista_objeto* suporta um único tipo genérico. Quando você declara objeto do tipo *lista_objeto*, precisa informar ao compilador qual tipo de dados essa ocorrência usará, como segue:

```
lista_objeto<char> lista_char;
lista_objeto<float> lista_float;
lista_objeto<personalizada> lista_personalizada;
```

Os comandos no fragmento de código a seguir criam três objetos *lista_objeto:* um do tipo *char*, um do tipo *float* e um do tipo *personalizado*. Na verdade, dadas as seguintes definições, seu objeto *lista_objeto* pode manter membros de lista *LivrosCoisas*:

```
class LivrosCoisa  {
  public:
    LivrosCoisa(char *titulo, char *editora, char *autor);
    void exibe_livro(void)
     {
       cout << "Livro: " << titulo << " por " << autor << " Editora:
       " << editora << endl;
     };
    operator char *(void);

  private:
    char titulo[64];
    char autor[64];
    char editora[64];
 };
// Código adicional do programa aqui

void main(void)
 {
   lista_objeto<LivrosCoisa> objeto1;
```

1193 COMPREENDENDO A CLASSE LISTA_LIGADA GENÉRICA

Na dica anterior, vimos que você deve preceder as definições de classe genérica com a palavra-chave *template* e um operador que corresponda ao tipo que o compilador deve implementar no tipo genérico. Quando você implementar a classe *lista_ligada*, que deriva da classe *lista_objeto*, também precisará fornecer as informações de tipo

genérico para o compilador. No entanto, quando você inicializar um objeto *lista_ligada* com um tipo específico, ele automaticamente inicializará todos os objetos *lista_objeto* que o objeto *lista_ligada* usa como o mesmo tipo específico. Você implementará a definição da classe *lista_ligada*, como mostrado aqui:

```
template <class DataT> class lista_ligada : public
          lista_objeto<DataT>  {
     lista_objeto<DataT> *inicio, *fim;
   public:
      lista_ligada(void)  {inicio = fim = NULL;}
   void armazena(DataT c);
   void remove(lista_objeto<DataT> *ob);
   void frentelista(void);
   void traslista(void);
   dbliga_ob<DataT> *localiza(DataT c);
   dbliga_ob<DataT> *pegainicio(void) {return inicio;}
   dbliga_ob<DataT> *pegafim(void)    {return fim;}
 };
```

Como você pode ver, tanto dentro da classe *lista_ligada*, que esta dica define, quanto dentro da classe *lista_objeto*, que a Dica 1192 define, a classe define cada função-membro e dados-membro em termos do objeto genérico *DataT*. Adicionalmente, as funções-membro que recebem um objeto de um dos dois tipos como um parâmetro (tal como a função-membro *remove* de *lista_ligada*) usa o objeto genérico *DataT* com a definição de parâmetro.

USANDO AS CLASSES GENÉRICAS COM UMA LISTA CHAR 1194

Na Dica 1190, você criou o programa *usa_liga.cpp*, que usou as classes originais *lista_objeto* e *lista_ligada* para manter uma pequena lista de variáveis de caractere. O CD-ROM que acompanha este livro inclui o programa *usa_gliga.cpp*, que usa as classes de lista genérica para criar uma lista *char* idêntica à lista criada pelo programa *usa_liga.cpp*.

Como você verá posteriormente, a única diferença significativa entre os programas *usa_liga.cpp* e *usa_gliga.cpp* é a seguinte declaração:

```
lista_ligada<char> lista;
char c;
lista_objeto<char> *p;
```

Como você sabe, a declaração da lista *lista_ligada<char>* cria o objeto *lista*. O objeto *lista* suporta *info* do tipo *char*. O restante do código dentro da função *main* é fundamentalmente o mesmo porque o programa *usa_gliga.cpp* sabe que está lidando com uma lista *char*, exatamente como *usa_liga.cpp* estava na Dica 1190. No entanto, na dica a seguir, você criará uma lista do tipo *double* para provar que suas classes genéricas trabalham com múltiplos tipos.

USANDO AS CLASSES GENÉRICAS COM UMA LISTA DUPLA 1195

Na dica anterior você usou as classes da lista genérica para criar uma lista com base em *char*. No entanto, como sabe, um benefício significativo das classes genéricas é que elas permitem que seus programas usem a mesma definição básica para criar múltiplos tipos de classes. O CD-ROM que acompanha este livro inclui o programa *dbl_liga.cpp*, que usa as classes genéricas para armazenar valores *double*.

O programa *dbl_liga.cpp* efetua processamento similar ao programa *usa_gliga.cpp* implementado na dica anterior. No entanto, as declarações diferem ligeiramente, o que causa impacto em todas as ações que o programa toma usando a lista, como segue:

```
lista_ligada<double> lista;
double c;
lista_objeto<double> *p;
```

Quando os comandos dentro da função *main* manipulam a lista, cada comando trata os valores *double*, em vez dos valores *char* usados na dica anterior.

1196 USANDO AS CLASSES GENÉRICAS COM UMA ESTRUTURA

Nas dicas anteriores, você criou listas duplamente ligadas simples com suas classes genéricas e tipos de dados simples. Como você descobrirá, o gabarito genérico requer trabalho extra antes de poder usar o gabarito com estruturas mais complexas. Para compreender por que algumas funções que trabalham com tipos simples são insuficientes para estruturas mais complexas, considere a seguinte função a partir da sua definição genérica existente:

```
friend ostream &operator<<(ostream &canal, lista_objeto<DataT> o)
 {
   canal << o.info << endl;
   return canal;
 }
```

A função de operador de inserção sobrecarregado sobrecarrega o canal de saída e coloca as informações que *info* armazena dentro do canal. Quando *info* é um tipo simples, esse comando é suficiente. No entanto, considere a seguinte definição de classe:

```
class Livro  {
   public:
     Livro(char *titulo, char *autor, char *editora, float preco);    //
     Construtora
     Livro(void)  {};
     void exibe_titulo(void);
     float pega_preco(void);
     void exibe(void);
     void atribui_editora(char *nome);
     bool operator==(Livro op2);
   private:
     char titulo[256];
     char autor[64];
     float preco;
     char editora[256];
     void exibe_editora(void);
 }
```

Como sabe, você não pode simplesmente inserir objetos de um tipo mais complexo (tal como o tipo *Livro*) em um canal de saída. Você precisa inserir membros individuais no canal de dados. No entanto, como o tipo genérico vê cada objeto inteiro dessa classe como o membro *info* da classe *lista_objeto*, você não pode simplesmente sobrecarregar o operador que *lista_objeto* sobrecarrega para *ostream*. Seus programas precisam, em vez disso, usar a identificação de tipo em tempo de execução para determinar qual tipo a lista atual usa e chamar a função explicitamente carregada apropriada para exibir os dados de forma correta. Alternativamente, você pode *forçar todas as classes que usam a lista a suportar uma função exibe ou outra função padrão, que seu programa* poderia então usar para exibir as informações a partir de dentro do objeto.

1197 SOBRECARREGANDO O OPERADOR DE COMPARAÇÃO ==

Como você aprendeu na dica anterior, se usar suas classes *lista_objeto* e *lista_ligada* com tipos de dados mais complexos, você precisa primeiro alterar algumas das definições gerais dentro de suas classes genéricas. Adicionalmente, suas classes componentes (que você armazenará dentro da lista) precisam sobrecarregar o operador de comparação ==. Como o método *localiza* examina um objeto e compara seu valor *info* a cada valor *info* na lista, e como *info* se refere a um objeto da classe, você não pode simplesmente usar o código tal como o seguinte dentro da classe genérica:

```
template <class DataT> lista_objeto<DataT>
         *lista_ligada<DataT>::
          localiza(lista_objeto<DataT> ob)
 {
   lista_objeto<DataT> *temp;
   temp = inicio;
   while(temp) {
```

```
        if(ob.info==temp->info)
            return temp;
        temp = temp ->pegaprox();
      }
    return NULL;
  }
```

Se você tentar usar o código como o que será mostrado no exemplo de código a seguir, o compilador gerará um erro, pois ele sabe que você não pode comparar explicitamente dois objetos como pode com dois tipos de dados simples. Portanto, quando suas listas genéricas usam tipos de dados complexos, você precisa garantir que seus tipos de dados complexos sobrecarregam o operador ==. Por exemplo, o fragmento de código a seguir sobrecarrega o operador == para a classe *Livro* que muitos dos seus programas usaram anteriormente:

```
bool Livro::operator==(Livro op2)
  {
    if(titulo != op2.titulo)
       return false;
    if(autor != op2.autor)
       return false;
    if(editora != op2.editora)
       return false;
    if(preco != op2.preco)
       return false;
    return true;
  }
```

A função do operador == sobrecarregado examina cada membro de dado dentro do objeto *Livro*. Se qualquer membro d.ferir do membro comparado, a comparação retornará *false* e encerrará a função. Se todos os membros forem o mesmo, a função retornará *true*. Observe que essa implementação em particular usa o tipo de dado *bool*, mas você poderia facilmente usar o tipo de dado *int*.

OUTRAS MELHORIAS NA LISTA GENÉRICA 1198

À medida que seus programas trabalharem mais com as funções de lista genérica, você provavelmente verá que existem várias implementações que pode acrescentar na lista ou modificar dentro das definições de lista básicas. Como as dicas anteriores identificaram, uma limitação significativa da classe *lista_ligada* é que a classe não classifica itens antes de inserir os itens dentro da lista. Você pode querer alterar a função básica *armazena* ou pode querer criar uma função *armazena_classifica*. Você pode até querer criar um gabarito de classe genérica classificada.

Adicionalmente, você pode querer que sua classe de lista (se ela não classifica itens automaticamente) possa armazenar informações dentro de vários locais na lista. Por exemplo, você poderia acrescentar as funções-membro *armazena_ini*, *armazena_fim* e *localiza_armazena* na sua classe. (A função-membro *localiza_armazena* encontra um item específico na lista e armazena as novas informações antes ou após o item encontrado). Alternativamente você pode querer adicionar um parâmetro inteiro na sua função-membro *armazena*, o que lhe permite chamar *armazena* com diferentes implementações. Finalmente, você pode querer modificar *armazena* para que ela receba uma ocorrência do objeto dos dados, em vez de um tipo de dados simples, e adicione o objeto na lista. A dica a seguir mostra como você poderia implementar uma função *armazena*, que recebe dados do objeto.

USANDO OBJETOS COM A FUNÇÃO ARMAZENA 1199

Como você aprendeu, as classes de ligação genéricas que criou, embora sejam um bom fundamento e úteis para gerenciar tipos de dados simples, têm alguns problemas quando você tenta armazenar tipos de dados complexos dentro da lista. Um dos problemas mais significativos com a construção atual é como ela insere informações em um novo *lista_objeto*. Por exemplo, com base em como você originalmente escreveu a função *armazena* e no que sabe das funções construtoras, você poderia tentar escrever a adição de um novo elemento à lista do tipo *Livro*, como mostrado aqui:

```
lista.armazena("Bíblia do Programador C/C++",
               "Jamsa e Klander", "Makron Books", 49.95);
```

Infelizmente, o compilador não reconhece a existência da função construtora e retornará um erro se você tentar criar novos elementos de lista da maneira mostrada no fragmento de código anterior. Além disso, o código é mais confuso que útil. O código a seguir, por outro lado, é mais claro — especialmente se seu programa processar as informações *Livro* antes de tentar armazenar as informações dentro de uma lista:

```
Livro cbib("Bíblia do Programador C/C++",
           "Jamsa e Klander", "Makron Books", 49.95);
// Código do programa aqui
lista.armazena(cbib);
```

No segundo fragmento de código, você primeiro cria o objeto, depois passa o objeto para a função *armazena*. Devido ao modo como você projetou a estrutura genérica, ela processa o objeto *cbib* perfeitamente e acrescenta-o à lista. O CD-ROM que acompanha este livro inclui o programa *bk_list.cpp*, que acrescenta três objetos à lista da maneira que o segundo fragmento de código nesta dica usa. Por causa da necessidade de retrabalhar o modo como a classe trata a saída, o programa não efetua todas as ações de saída que os programas de lista nas dicas anteriores efetuaram. No entanto, o programa *bk_list.cpp* percorre a lista um item de cada vez e gera saída. Quando você compilar e executar o programa *bk_list.cpp*, sua tela exibirá a seguinte saída:

```
Aqui estão alguns itens.
Percorre a lista 'manualmente'.
Título: Bíblia do Programador C/C++
Editora: Makron Books
Título: 1001 Dicas em Visual Basic
Editora: Makron Books
Título: Hacker Proof
Editora: Jamsa Press
C:\>
```

1200 ESCREVENDO UMA FUNÇÃO PARA DETERMINAR O COMPRIMENTO DA LISTA

Na Dica 1198, você aprendeu sobre as melhorias que seus programas podem fazer nas classes de lista genérica. Uma melhoria que a Dica 1198 discutiu é a adição de uma função que percorre a lista e retorna um contador do número de itens dentro da lista. O fragmento de código a seguir fornece uma implementação de exemplo da função-membro *listatam* que conta itens:

```
template <class  DataT> int lista_ligada<DataT>::listatam(void)
 {
   lista_objeto<DataT> *temp;
   int = conta = 0;

   temp = inicio;
   do  {
         temp = temp->pega prox();
         conta = conta + 1;
   }  while (temp);
   cout << "Número de itens na lista: " << cout << endl;
   return conta;
 }
```

Como você pode ver, a função-membro *listatam* executa basicamente o mesmo processamento que a função *frentelista*, exceto que mantém um contador à medida que percorre a lista. A função-membro *listatam* então exibe o contador no final de seu processo e retorna o contador ao *lvalue* dentro do comando chamador. O CD-ROM que acompanha este livro inclui o programa *cnt_lst.cpp*, que executa o mesmo processamento que o programa *usa_gliga.cpp* apresentado na Dica 1194. No entanto, o programa *cnt_lst.cpp* também chama *listatam* em diferentes pontos dentro da execução do programa.

INTRODUZINDO A BIBLIOTECA DE GABARITOS PADRÕES 1201

A Biblioteca de Gabaritos Padrões, ou STL, é uma biblioteca de classes contêiner de C++ (tais como listas ligadas), algoritmos e iteradores; ela fornece muitos dos algoritmos básicos e estruturas de dados da ciência da computação (tais como classificação, mapeamento e funções matemáticas). A Biblioteca de Gabaritos Padrões é uma biblioteca genérica, o que significa que seus projetistas parametrizaram bastante os componentes da biblioteca: quase todo componente nela é um gabarito. Você deve se certificar de que compreende como os gabaritos trabalham em C++ antes de usar a Biblioteca de Gabaritos Padrões. Você pode usar a Biblioteca de Gabaritos Padrões rapidamente e facilmente:

- criar listas classificadas dos objetos
- criar listas classificadas de objetos relacionados que compartilham uma única chave
- manipular estruturas de dados complexas de um modo simples
- executar manipulações complexas das informações armazenadas dentro de um repositório usando algoritmos predefinidos.

Nas próximas cinquênta dicas, você usará os componentes da Biblioteca de Gabaritos Padrões para escrever vários programas. Antes de iniciar, deverá observar os seguintes pontos importantes:

1. Como o compilador *Turbo C++ Lite* não suporta as definições genéricas, você não pode usar a Biblioteca de Gabaritos Padrões com ele.
2. Tanto o *Visual C++* quanto o Borland *C++ 5.02 for Windows* incluem os arquivos de cabeçalho para a maior parte da Biblioteca de Gabaritos Padrões, o que significa que você pode usar muitos recursos da Biblioteca de Gabaritos Padrões com qualquer um desses compiladores sem outras modificações.
3. Se seu compilador não inclui a Biblioteca de Gabaritos Padrões, você pode descarregar os arquivos de cabeçalho necessários. Para descarregar os arquivos da Biblioteca de Gabaritos Padrões, visite o site da Silicon Graphics na Web em *http://www.sgi.com/Technology/STL/index.html*, como mostrado na Figura 1201.

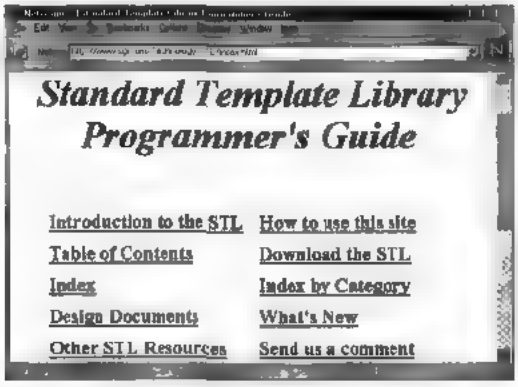

Figura 1201 A página de apresentação da Silicon Graphics STL.

COMPREENDENDO OS ARQUIVOS DE CABEÇALHO DA BIBLIOTECA DE GABARITOS PADRÕES 1202

Como você aprendeu na dica anterior, a Biblioteca de Gabaritos Padrões fornece classes e funções genéricas que seus programas podem usar para criar funcionalidade adicional. Devido ao número de classes que a Biblioteca de Gabaritos Padrões fornece, os projetistas dividiram a biblioteca em múltiplos arquivos de cabeçalho para reduzir o tempo de compilação. A Tabela 1202 detalha os arquivos de cabeçalho.

Tabela 1202 Os arquivos de inclusão para a biblioteca de gabaritos padrões.

Nome do Arquivo	Descrição
algo.h	Inclui todos os algoritmos da Biblioteca de Gabaritos Padrões. Dicas posteriores detalharão os algoritmos da Biblioteca de Gabaritos Padrões (também chamados algorith.h em algumas implementações da STL).
bool.h	Define o tipo de dados *bool*.
bvector.h	Define os objetos *bit_vector* que seus programas podem usar para manter estruturas como matrizes de dados bit.
deque.h	Define objetos *deque* que seus programas podem usar para criar estruturas similares a estruturas que você pode manipular no início e no final.
function.h	Inclui os operadores, objetos de função e adaptadores de função que gerenciam as classes da Biblioteca de Gabaritos Padrões.
iterator.h	Define os tags de iterador, iteradores de canal e adaptadores de iterador para as classes da Biblioteca de Gabaritos Padrões.
list.h	Define um objeto de lista duplamente ligada.
map.h	Define a classe *map*, uma lista duplamente ligada classificada com um valor-chave e um valor de dados.
multimap.h	Define a classe *multimap*, uma lista classificada duplamente ligada com um ou mais valores-chave e um valor de dados. Os objetos *multimap* suportam um ou mais campos classificados dentro da lista.
multiset.h	Define a classe *multiset*, que permite que seus programas representem listas ligadas classificadas de um modo que permite a consulta, inserção e remoção de um elemento arbitrário com um número de operações proporcionais com o logaritmo do número de elementos na seqüência (um valor conhecido como *tempo logarítmico*). Ao contrário da classe *set*, *multisets* podem usar múltiplas chaves de classificação.
pair.h	Define a classe *pair*, que permite que seus programas armazenem dois valores (do mesmo tipo ou tipos diferentes) dentro de um único objeto.
random.c	Define um gerador de números aleatórios. Você deve incluir *random.c* caso seus programas usem o algoritmo *random_shuffle*.
set.h	Define a classe *set*, que permite que seus programas representem listas ligadas classificadas em um modo que permite a consulta, inserção e remoção de um elemento arbitrário em tempo logarítmico. Ao contrário da classe *multiset*, os *sets* precisam usar somente uma única chave de classificação.
stack.h	Define um objeto *stack* que seus programas podem usar para controlar uma seqüência variável de elementos. O objeto aloca e libera armazenagem para a seqüência que controla.
tempbuf.c	Um arquivo de programa para suportar um buffer auxiliar para os algoritmos *get_temporary_buffer*, *stable_partition*, *inplace_merge* e *stable_sort*. O arquivo de cabeçalho *algo.h* inclui automaticamente tanto *tempbuf.c* quanto *tempbuf.h*. Você nunca deve adicionar esse arquivo aos seus programas diretamente.
tempbuf.h	Contém protótipos de função e definições de classe para o arquivo de programa *tempbuf.c*.
vector.h	Define a classe de gabarito *vector*, um objeto que controla uma seqüência de tamanho variável de elementos. Ao contrário das classes *list*, *map* e *set*, a classe *vector* é uma lista ligada simples que o compilador trata como uma matriz.

Nota: Existem alguns outros arquivos de inclusão da Biblioteca de Gabaritos Padrões além desses, mas eles são principalmente para uso com compiladores C++ específicos para o DOS/Windows que não suportam automaticamente múltiplos modelos de memória. Você não irá requerer esses outros arquivos de inclusão se usar o Visual C++ ou o Borland C++ 5.02 for Windows. No entanto, se usar um compilador diferente, verifique a documentação do compilador para determinar se ele requer arquivos de inclusão adicionais.

COMPREENDENDO OS REPOSITÓRIOS 1203

Um dos blocos de construção fundamentais da Biblioteca de Gabaritos Padrões é o *repositório* (ou *container*). O repositório é um objeto que armazena coleções de outros objetos (similarmente tipados). Por exemplo, a classe *lista_ligada* que você criou na Dica 1190 é um repositório para objetos da classe *lista_objeto*. A Figura 1203.1 ilustra um modelo de um repositório e os objetos que o repositório armazena dentro de si mesmo.

Figura 1203.1 O modelo lógico para um repositório e os objetos que ele contém.

À medida que você continuar a escrever programas mais complexos, verá que usar repositórios (seja a partir da Biblioteca de Gabaritos Padrões, de outras bibliotecas, ou do seu próprio projeto) torna-se mais importante para seus programas. Em vez de tentar manipular grandes números de objetos individuais, que requerem um número similar de variáveis, laços, testes e assim por diante, você pode em vez disso incluir muitos objetos dentro de um único repositório, o que simplifica o gerenciamento dos objetos. Por exemplo, é muito mais fácil gerenciar uma matriz de 10 inteiros que gerenciar 10 variáveis inteiras. Na verdade, muitos programas que você escreveu até aqui usaram o tipo mais simples de repositório: uma matriz. A Figura 1203.2 ilustra uma matriz como um repositório.

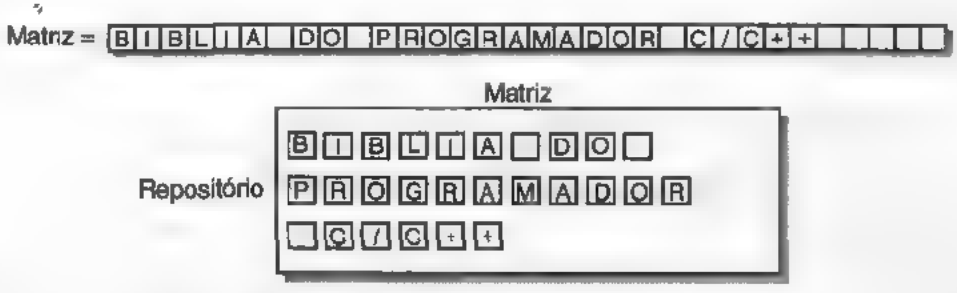

Figura 1203.2 A matriz como um repositório.

Em dicas posteriores você aprenderá mais sobre os repositórios e implementará a Biblioteca de Gabaritos Padrões. No entanto, na Dica a seguir, você usará as classes *lista_ligada* e *lista_objeto* para revisar o conceito dos repositórios.

USANDO UM EXEMPLO DE REPOSITÓRIO 1204

Na dica anterior você aprendeu sobre os repositórios e como seus programas usarão classes repositório para manter as informações sobre os grupos como objetos. Como você aprendeu na Dica 1201, a Biblioteca de Gabaritos Padrões inclui vários tipos de repositórios diferentes (incluindo *lists*, *sets*, *maps* e *deques*). Na dica a seguir, você aprenderá os fundamentos sobre os tipos básicos da Biblioteca de Gabaritos Padrões.

No entanto, antes de continuar a aprender sobre os repositórios da Biblioteca de Gabaritos Padrões, é importante analisar uma classe genérica que você criou anteriormente para determinar se ela qualifica como um re-

positório. Em dicas anteriores você projetou e implementou a classe *lista_ligada*, que definiu genericamente como mostrado aqui:

```
template <class DataT> class lista_ligada : public lista_objeto<DataT>
 {
    lista_objeto<DataT> *inicio, *fim;
  public :
    lista_ligada(void) {inicio = fim = NULL;}
    void armazena(DataT c);
    void remove(lista_objeto<DataT> *ob);
    void frentelista(void);
    void traslista(void);
    lista_objeto<DataT> *encontra<DataT c);
    lista_objeto<DataT> *pegainicio(void) {return inicio;}
    lista_objeto<DataT> *pegafim(void) {return fim;}
 };
```

A classe *lista_ligada*, logicamente, derivou da classe *lista_objeto*, que você definiu como a seguir (por clareza, essa reimpressão realoca as funções amigas in-line fora da definição da classe):

```
template <class DataT> class lista_objeto
 {
   public:
     DataT info;
     lista_objeto<DataT> *proximo;
     lista_objeto(DataT *anterior;
     lista_objeto(void)
     {
        info = 0;
        proximo = NULL;
        anterior = NULL;
     }
     lista_objeto(DataT c)
     {
        info = 0;
        proximo = NULL;
        anterior = NULL;
     }
     lista_objeto<DataT> *pegaproximo(void) { return proximo; }
     lista_objeto<DataT> *pegaanterior(void) { return anterior; }
     void pegainfo(DataT &c) { c = info; }
     void muda(DataT c)  {info = c;}
     friend ostream &operator<<(ostream &canal, lista_objeto<DataT> o)
     friend ostream &operator<<(ostream &canal, lista_objeto<DataT> *o)
     friend istream &operator>>(ostream &canal, lista_objeto<DataT> *o)
 };
```

Como a classe *lista_objeto* é genérica, ela aceitará dados de qualquer tipo. Cada uma das declarações a seguir é válida:

```
lista_objeto<int> int_obj;
lista_objeto<float> float_obj;
lista_objeto<char> char_obj;
lista_objeto<Livro> livro_obj;
```

A definição genérica da classe significa que esse tipo não é importante (em outras palavras, seus programas podem facilmente definir listas *int* e listas *float*, e as funções de classe-membro *lista* funcionarão igualmente bem em ambas). Como você sabe, um repositório gerencia objetos individuais de tipos conhecidos e desconhecidos sem preocupação com o tipo do objeto. Por exemplo, cada uma das seguintes declarações de matriz é válida:

```
int intmatriz[10];
float floatmatriz[10];
```

```
char charmatriz[10];
Livro livromatriz[10];
```

Embora as informações que cada elemento dentro das matrizes armazena sejam diferentes, as matrizes propriamente são idênticas. Isto é, cada matriz contém dez elementos de algum tipo, que o usuário pode percorrer utilizando o índice da matriz.

A classe *lista_ligada* é similar a uma matriz. Como ela deriva do tipo *lista_objeto*, o compilador tipa explicitamente cada ocorrência da lista (em outras palavras, se os *lista_objetos* são todos *doubles*, o objeto *lista_ligada* precisa também ser do tipo *double*). No entanto, a classe *lista_ligada* não está preocupada com a natureza dos objetos que armazena; ela simplesmente armazena esses objetos e oferece a você modos de navegar os objetos dentro da *lista_ligada*. Em outras palavras, a classe *lista_ligada* é claramente um repositório.

APRESENTANDO OS REPOSITÓRIOS DA BIBLIOTECA DE GABARITOS PADRÕES 1205

Como você aprendeu, o repositório é um dos blocos de construção fundamentais da Biblioteca de Gabaritos Padrões. Na dica anterior, você analisou como projetou e implementou anteriormente muitos repositórios dentro de seus programas, mesmo quando você nem percebeu que eles eram repositórios. Como aprenderá, a Biblioteca de Gabaritos Padrões define implementações similares ou relacionadas a muitos dos repositórios que você usou anteriormente, bem como alguns repositórios que pode nem ter considerado. A Biblioteca de Gabaritos Padrões suporta dois tipos básicos de repositórios: *Repositórios de Seqüência* e *Repositórios Associativos*. Os Repositórios de Sequência são objetos que armazenam coleções de outros objetos em um arranjo estritamente linear. A Biblioteca de Gabaritos Padrões suporta os seguintes três Repositórios de Seqüência:

- *vector<T>*: A classe *vector<T>* fornece acesso aleatório da forma de matrizes para uma seqüência de objetos. À medida que seu programa é executado, o tamanho do objeto *vector* pode variar. Seus programas podem efetuar inserções e exclusões no final da seqüência. Você tipicamente usa vetores para manter informações não-classificadas em uma série.
- *deque<T>*: A classe *deque<T>* fornece acesso aleatório a uma seqüência de objetos. Como com os *vetores*, à medida que seu programa é executado, o tamanho do objeto *deque* pode variar. Seus programas podem efetuar inserções e exclusões tanto no início quanto no final da seqüência. Você geralmente usará *deques* para manter informações não-classificadas em uma série quando não tiver certeza em qual final da série você acrescentará informações.
- *list<T>*: A classe *list<T>* fornece acesso a uma seqüência de objetos. Como com *vectors* e *deques*, à medida que seu programa for executado, o tamanho do objeto *list* poderá variar. Seus programas podem efetuar inserções e exclusões em qualquer ponto da sequência. Você geralmente usará *lists* para manter informações não-classificadas em uma série quando não tiver certeza de onde na série inserirá informações.

Por outro lado, os Repositórios Associativos oferecem ao seu programa um modo fácil de recuperar objetos rapidamente a partir da coleção de objetos que a classe repositório contém. Os Repositórios Associativos usam chaves para suportar a recuperação rápida dos objetos. O tamanho da coleção pode variar em tempo de execução (como podem todos os repositórios da Biblioteca de Gabaritos Padrões, tornando-os *repositórios dinâmicos*, enquanto que as matrizes e outros repositórios simples são repositórios *estáticos*). Um Repositório Associativo mantém a coleção na ordem, com base em um objeto de função de comparação do tipo *Compare*. A Biblioteca de Gabaritos Padrões suporta os seguintes quatro Repositórios Associativos:

- *set<T, Compare>*: A classe *set* suporta chaves únicas (isto é, os objetos da classe contêm, no máximo, um de cada valor-chave) e fornece recuperação rápida das chaves. Você geralmente usará *sets* para manter informações classificadas que usam somente uma única chave de classificação, tal como uma série simples de números inteiros.
- *multiset<T, Compare>*: A classe *multiset* suporta chaves duplicadas (isto é, os objetos da classe possivelmente contêm múltiplas cópias do mesmo valor-chave) e oferece a recuperação rápida das próprias chaves. Você geralmente usará *multusets* para manter informações classificadas que usam mais de uma chave de classificação, que também são os dados, tal como uma série classificada de coordenadas em uma grade.

- *map<Tecla, T, Compare>:* A classe *map* suporta chaves únicas (isto é, objetos da classe contém, no máximo, um de cada valor-chave) e fornece recuperação rápida de outro tipo *T* com base nas chaves. Você geralmente usará *maps* para manter informações classificadas que usam somente uma única chave classificada, tal como um banco de dados simples de números telefônicos.
- *multimap<Chave, T, Compare>:* A classe *multimap* suporta chaves duplicadas (isto é, os objetos da classe possivelmente contêm múltiplas cópias do mesmo valor-chave) e permite a rápida recuperação de outro tipo *T* com base nas chaves. Você geralmente usará *multimaps* para manter informações classificadas que usam múltiplas chaves de classificação, tal como um banco de dados complexo de clientes.

1206 COMPREENDENDO OS REPOSITÓRIOS FORWARD E REVERSIBLE

Vimos na dica anterior que os seus programas que usam a Biblioteca de Gabaritos Padrões podem usar dois tipos básicos de repositórios: Repositórios de Seqüência e Repositórios Associativos. A Biblioteca de Gabaritos Padrões deriva ambos esses tipos específicos de repositórios a partir de dois tipos de repositórios mais gerais: *Repositórios Forward* e *Repositórios Reversible*. Essencialmente, você pode pensar nos Repositórios Forward como listas ligadas simples. Os repositórios de seqüência derivam dos Repositórios Forward. Seus programas somente podem negociar Repositórios Forward em uma única direção. A Figura 1206.1 mostra como seus programas podem acessar os Repositórios Forward.

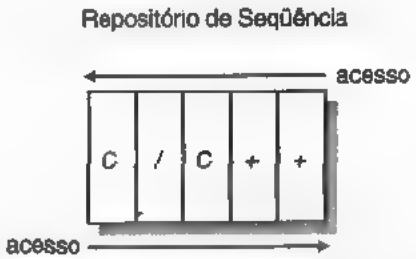

Figura 1206.1 Seus programas podem acessar os Repositórios Forward em uma única direção.

Os Repositórios Reversíveis, por outro lado, derivam das listas ligadas duplas. Os Repositórios Reversíveis fornecem meios fáceis para os programas negociarem uma lista para a frente ou de trás para a frente, para os programas obterem facilmente o início ou o final da lista e assim por diante. Seus programas criarão Repositórios Reversíveis dinamicamente. A Biblioteca de Gabaritos Padrões derivam os Repositórios Associativos tanto a partir dos Repositórios Forward quanto dos Repositórios Associativos. A Figura 1206.2 mostra como seus programas podem acessar os Repositórios Associativos.

Figura 1206.2 Seu programa pode acessar Repositórios Associativos em qualquer ponto no repositório.

COMPREENDENDO OS REPOSITÓRIOS DE SEQÜÊNCIA DA BIBLIOTECA DE GABARITOS PADRÕES 1207

Como você aprendeu na Dica 1205, um Repositório de Sequência é um objeto que armazena coleções de outros objetos em um arranjo estritamente linear. A Biblioteca de Gabaritos Padrões suporta três Repositórios de Seqüência: *vectors*, *deques* e *lists*, bem como dois gabaritos derivativos, *bit_vectors* e *slists*. Para compreender como seus programas usarão os Repositórios de Sequência, considere o seguinte programa, *vetor1.cpp*, que usa o arquivo de cabeçalho *vector.h* para criar um vetor vazio de inteiros, e depois manipular esse vetor:

```
//  Se você usa o Visual C++, defina as opções do
//  compilador como /GX
#include <iostream.h>
#include <vector.h>

using namespace std;
typedef vector<int> INTVECTOR;
const TAM_MATRIZ = 4;

void main(void)
 {
   // O vetor alocado dinamicamente inicia com 0 elementos.
   INTVECTOR vetor;

   // Inicializa a matriz para conter os membros
   // [100, 200, 300, 400]
   for (int cCadaItem = 0; cCadaItem < TAM_MATRIZ; cCadaItem++)
      vetor.push_back((cCadaItem + 1) * 100);

   cout << "Primeiro elemento: " << vetor.front() << endl;
   cout << "Último elemento: " << vetor.back() << endl;
   cout << "Elementos no vetor: " << vetor.size() << endl;

   // Exclua o último elemento do vetor. Lembre-se de que o
   // vetor inicia em 0, de modo que Vetor.end() na
   // verdade aponta para 1 elemento após o final.
   cout << "Excluindo o último elemento." << endl;
   vetor.erase(vetor.end() - 1);
   cout << "O novo elemento é: " << vetor.back() << endl;

   // Exclua o primeiro elemento do vetor.
   cout << "Excluindo o primeiro elemento." << endl;
   vetor.erase(vetor.begin());
   cout << "O novo elemento é: " << vetor.front() << endl;
   cout << "Elementos no vetor: " << vetor.size() << endl;
 }
```

O programa *vetor1.cpp* declara um vetor vazio de inteiros, depois inicializa o vetor com os membros [100, 200, 300 e 400]. Em seguida, o programa usa a função-membro *vetor.front* para obter e exibir o primeiro elemento do vetor. Após exibir o primeiro elemento, o programa usa a funçao-membro *vetor.back* para obter e exibir o último elemento do vetor. Após exibir o primeiro elemento do vetor, o programa usa a função-membro *vetor.back* para obter e exibir o último elemento do vetor. O programa também usa a função-membro *vetor.size* para exibir o número de elementos no vetor. Após exibir o número de elementos no vetor, o programa usa a função-membro *vetor.end* para mover para além do final do vetor, subtrai 1 dessa posição e usa a função-membro *vetor erase* para apagar o último elemento. Após apagar o último elemento, o programa usa a função-membro *vetor.back* para exibir o novo último elemento. Após exibir o novo último elemento, o programa usa a função-membro *vetor.erase*, juntamente com a função-membro *vetor.begin*, para apagar o primeiro elemento do vetor, e depois usa *vetor.front* para exibir o primeiro elemento. Finalmente, o programa usa a função-membro *vetor.size* para exibir o número de elementos restantes no vetor. Quando você compilar e executar o programa *vetor1.cpp*, sua tela exibirá a seguinte saída:

```
Primeiro elemento: 100
Último elemento: 400
Elementos no vetor: 4

Excluindo o último elemento.
O novo último elemento é: 300
Excluindo o primeiro elemento.
O novo primeiro elemento é: 200
Elementos no vetor: 2
C:\>
```

1208 COMPREENDENDO O COMANDO USING NAMESPACE STD

Na dica anterior, você escreveu o programa *vetor1.cpp*, que criou um repositório de vetor simples e manipulou seus objetos componentes. Embora a maior parte do programa seja auto-explicativa, a seguinte linha de código pode parecer desnecessária ou fora de lugar para você:

```
using namespace std;
```

Como você sabe, o comando *using* permite que seu programa acesse nomes de variáveis a partir de dentro de um determinado *namespace*. Neste caso em particular, o comando *using* permite que seu programa acesse as variáveis e classes dentro do *namespace std*, que é o *namespace* padrão para a Biblioteca de Gabaritos Padrões. Sempre que você escrever programas que usam componentes da Biblioteca de Gabaritos Padrões, precisa incluir o comando *using namespace std* ou o compilador não reconhecerá a classe ou classes da Biblioteca de Gabaritos Padrões que seu programa usa.

1209 COMPREENDENDO OS REPOSITÓRIOS ASSOCIATIVOS DA BIBLIOTECA DE GABARITOS PADRÕES

Um Repositório Associativo é um repositório de tamanho variável que suporta a recuperação eficiente de elemento (valor) com base em *chaves*. Uma chave é um valor de classificação que pode ou não ser o valor real que o repositório usa como um índice para os objetos do repositório. Os Repositórios Associativos suportam inserção e remoção de elementos, mas diferem dos Repositórios de Seqüência em que os Repositórios Associativos não fornecem um mecanismo para inserir um elemento em uma determinada posição. Como com todos os repositórios, os elementos no Repositório Associativo têm tipo *valor_tipo*. Adicionalmente, cada elemento no Repositório Associativo tem uma chave do tipo *chave_tipo*.

Em alguns Repositórios Associativos, tais como os *Repositórios Associativos Simples* (*sets* e *multisets*), o *valor_tipo* e *chave_tipo* são o mesmo — isto é, os elementos são suas próprias chaves. Em outros, a chave é alguma parte específica do valor. Como os Repositórios Associativos armazenam elementos de acordo com suas chaves, é essencial que a chave que o repositório associe com cada elemento seja imutável (isto é, o programa não pode alterar o elemento). Portanto, nos Repositórios Associativos Simples, os elementos são imutáveis. Em outros tipos de Repositório Associativo, tais como os Repositórios Associativos Pares, os elementos são mutáveis, mas o programa não pode modificar a parte de um elemento que é a chave do elemento.

Nos Repositórios Associativos Simples, onde os elementos são as chaves, os elementos são completamente imutáveis. Portanto, os tipos membros *iterator* e *const_iterator* são os mesmos para Repositórios Associativos Simples. No entanto, outros tipos de Repositórios Associativos têm elementos mutáveis e fornecem iteradores por meio dos quais os programas podem modificar os elementos.

Em alguns Repositórios Associativos, tais como os Repositórios Associativos Únicos, a especificação da classe da Biblioteca de Gabaritos Padrões garante que dois elementos não têm a mesma chave. Em outros Repositórios Associativos, tais como Repositórios Associativos Múltiplos, o repositório permitirá que o programa armazene múltiplos elementos com a mesma chave dentro do repositório. Para compreender melhor os Repositórios Associativos, considere o programa a seguir, *primset.cpp*, que cria e manipula um objeto do tipo *set*:

```
//  Se você usar o Visual C++, defina as opções do
//  compilador como /GX
#include <iostream.h>
#include <set.h>
```

```
using namespace std;
typedef set<int> SET_INT;

void main(void)
 {
   SET_INT s1;
   SET_INT s2;
   SET_INT::iterator i;

   cout << "s1.insert(5)" << endl;
   s1.insert(5);
   cout << "s1.insert(10)" << endl;
   s1.insert(10);
   cout << "s1.insert(15)" << endl;
   s1.insert(15);
   cout << "s2.insert(2)" << endl;
   s2.insert(2);
   cout << "s2.insert(4)" << endl;
   s2.insert(4);
  cout << "swap(s1,s2)" << endl;
  swap(s1,s2);
  for (i = s1.begin();i != s1.end();i++)            // Exibe: 2,4
     cout << "s1 tem " << *i << " em seu set." << endl;

  for (i = s2.begin();i != s2.end();i++)       // Exibe: 5,10,15
     cout << "s2 tem " << *i << " em seu set." << endl;

  cout << "s1.swap(s2)" << endl;
  s1.swap(s2);
  for (i = s1.begin();i != s1.end();i++)       // Exibe: 5,10,15
     cout << "s1 tem " << *i << " em seu set." << endl;
  for (i = s2.begin();i != s2.end();i++)       // Exibe: 2,4
     cout << "s2 tem " << *i << " em seu set." << endl;
 }
```

O programa cria dois *sets*, atribuindo ao primeiro os valores 5, 10 e 15, e, ao segundo, os valores 2 e 4. O programa também define o iterador *i*, que ele usa posteriormente para percorrer os *sets*. Dentro do programa *primset.cpp*, o código usa três funções-membro da classe *set*. A função *swap* permuta as duas seqüências controladas. A função *begin* retorna um *iterador bidirecional* que aponta para o primeiro elemento da seqüência. A função *end* retorna o iterador bidirecional que aponta para além do final da seqüência. (Você aprenderá mais sobre os iteradores na dica a seguir.) O programa insere elementos no conjunto, depois percorre o conjunto permutando os elementos. Quando você compilar e executar o programa *primset.cpp*, sua tela exibirá a seguinte saída:

```
s1.insert(5)
s1.insert(10)
s1.insert(15)
s2.insert(2)
s2.insert(4)
swap(s1, s2)
s1 tem 2 em seu set.
s1 tem 4 em seu set.
s2 tem 5 em seu set.
s2 tem 10 em seu set.
s2 tem 15 em seu set.
s1.swap(s2)
s1 tem 5 em seu set.
s1 tem 10 em seu set.
s1 tem 15 em seu set.
s2 tem 2 em seu set.
s2 tem 4 em seu set.
C:\>
```

COMPREENDENDO OS ITERADORES 1210

Um fator-chave no projeto da Biblioteca de Gabaritos Padrões é o uso consistente de iteradores nas definições do repositório, que generalizam os ponteiros C++ como intermediários entre algoritmos e repositórios. A Biblioteca de Gabaritos Padrões define cinco categorias de iteradores. A classificação é também o guia principal para estender a biblioteca para incluir novos algoritmos que trabalham com os repositórios da Biblioteca de Gabaritos Padrões, ou incluam novos repositórios para os quais você possa aplicar muitos algoritmos genéricos da Biblioteca de Gabaritos Padrões.

Existem três fatos que você precisa levar em conta ao tentar determinar quais algoritmos pode usar com quais repositórios e iteradores:

- A Biblioteca de Gabaritos Padrões classifica os iteradores em cinco categorias: *forward, entrada, saída, bidirecional* e *acesso aleatório.*
- Cada descrição de classe de repositório inclui os tipos da categoria iterador que a classe repositório oferece.
- Cada descrição de algoritmo genérico inclui as categorias iterator e repositório com o qual o algoritmo genérico trabalha.

A Tabela 1210.1 define as cinco categorias de tipo iterador.

Tabela 1210 Os cinco tipos de iteradores de Biblioteca de Gabaritos Padrões.

Tipo de Iterador	Descrição
forward	Permite que uma seqüência seja percorrida em uma direção, que seus programas expressarão com o operador de incremento (++).
entrada	Similar aos iteradores *forward* em que seus programas podem usá-los para inserir dados em um repositório. No entanto, os iteradores de *entrada* não suportam todas as propriedades dos iteradores *forward.*
saída	Similar aos iteradores *forward* em que seus programas podem usá-los para escrever dados a partir de um repositório. No entanto, os iteradores de *saída* podem não suportar todas as propriedades dos iteradores *forward.*
bidirecional	Permite percorrer em ambas as direções, que seus programas expressarão com ++ (para a frente) e -- (para trás). Observe que, se o iterador aponta a partir do final do repositório para a frente, incrementar o iterador com ++ na verdade moverá o iterador em direção ao início do repositório.
acesso aleatório	Permite que uma seqüência possa ser percorrida na forma bidirecional. Adicionalmente, os iteradores de acesso aleatório lhe permitem "saltos longos" bidirecionais dentro de uma seqüência, o que você expressará como adição inteira, subtração inteira, subtração de iterador ou comparações, como detalha a Tabela 1210.2.

Como indica a Tabela 1210.1, os iteradores de acesso aleatório permitem usar várias técnicas diferentes para percorrer as seqüências. A Tabela 1210.2 detalha as técnicas que seus programas podem usar com os iteradores de *acesso aleatório* para percorrer uma lista.

Tabela 1210.2 As atividades que seus programas podem efetuar nos iteradores de ***acesso aleatório***.

Técnica	Explicação
adição inteira	Você pode efetuar adição e subtração inteira para um iterador de *acesso aleatório* usando as formas `r += n` e `r -= n` (onde `r` é um iterador de acesso aleatório e `n` é um inteiro). A operação produz um resultado iterador.
adição e subtração	Você pode adicionar ou subtrair um inteiro de um iterador usando as formas `r + n` e `r - n` (onde `r` é um iterador de acesso aleatório e `n` é um inteiro). A operação produz um resultado iterador.

Tabela 1210.2 As atividades que seus programas podem efetuar nos iteradores de acesso aleatório. (*Continuação*)

Técnica	Explicação
subtração de iterador	Você pode subtrair um iterador de um iterador usando a forma *r - s* (onde *r* é um iterador de acesso aleatório e *s* é outro iterador de *acesso aleatório*). A operação produz um resultado iterador.
comparações	Você pode efetuar comparações com iteradores de acesso aleatório usando as formas *r<s*, *r>s*, *r<=s* e *r>=s*. As comparações de iterador produzem valores *booleanos*.

Além disso, todas as cinco categorias de iteradores também permitem as seguintes ações:

- Testar a igualdade com o operador == e a desigualdade com o operador !=.
- Desreferenciar, o que significa obter o objeto de dados na posição que o iterador referencia, o que é expresso com * (o operador de desreferenciar ponteiro).

No entanto, a Biblioteca de Gabaritos Padrões não garante que as ações a seguir executarão como você espera quando seus programas manipularem os iteradores:

- A Biblioteca de Gabaritos Padrões não garante que você possa salvar um iterador de *entrada* ou *saída*, e *use-o para começar a avançar de sua posição atual posteriormente*.
- A Biblioteca de Gabaritos Padrões não garante que você possa mais tarde atribuir um valor a um objeto que você obteve anteriormente de um repositório aplicando o operador de redireçao de ponteiro (*) a um iterador de *entrada*.
- A Biblioteca de Gabaritos Padrões não garante que você possa ler a partir de um objeto, que obteve a partir de um repositório aplicando * a um iterador de *saída*.
- A Biblioteca de Gabaritos Padrões não garante que você possa testar a igualdade ou desigualdade de dois iteradores de saída (isto é, == e != não podem ser definidos quando você os aplica em iteradores de saída).

USANDO UM EXEMPLO DE ITERADOR 1211

Como você aprendeu na dica anterior, seus programas usarão iteradores para percorrer ou para manter suas posições dentro de um objeto repositório. Os diferentes tipos de repositório suportam diferentes tipos de iteradores, como detalha a Tabela 1211.

Tabela 1211 Os tipos de iteradores que cada repositório usa.

Tipo de Repositório	Tipo de Iterador
vector<T>::iterador	Iterador de *acesso aleatório*
deque<T>::iterador	Iterador de *acesso aleatório*
list<T>::iterador	Iterador *bidirecional*
repositórios associativos	Todos os repositórios associativos usam *iteradores bidirecionais*.

Como detalha a Tabela 1211, o tipo *list* usa um iterador *bidirecional*. O programa a seguir, *usa_iter.cpp*, usa um iterador *bidirecional* para percorrer uma lista de inteiros:

```
//  Se você usa o Visual C++, defina as opções do
//  compilador como /GX
#include <iostream.h>
#include <list.h>

using namespace std;
typedef list<int> LISTINT;

void main(void)
 {
   LISTINT listaUm;
   LISTINT::iterator i;
   // Acrescenta alguns dados
   listaUm.push_front (2);
```

```
    listaUm.push_front (1);
    listaUm.push_back (3);

    for (i = listaUm.begin(); i != listaUm.end(); ++i) // list values 1 2 3
      cout << *i << " ";
    cout << endl;
    for (i = listaUm.end(); i != listaUm.begin(); --i)   // list values 1 1 1 1
      cout << *i << " ";
    cout << endl;

}
```

O programa *usa_iter.cpp* cria primeiro o tipo *LISTINT* e, depois, cria a ocorrência *listaUm* do tipo. Em seguida, o programa usa os métodos *push_front* e *push_back* para acrescentar três valores na lista (dois para a frente e um para trás). O programa então usa o iterador *i* para percorrer a lista para a frente e para trás. Quando você compilar e executar o programa *usa_iter.cpp*, sua tela exibirá a seguinte saída (observe que o primeiro valor é um 0 porque a função-membro *end* obtém a localização logo após o final do vetor):

```
1 2 3
0 3 2
C:\>
```

1212 COMPREENDENDO MELHOR OS TIPOS DE ITERADOR DE ENTRADA E SAÍDA DA BIBLIOTECA DE GABARITOS PADRÕES (STL)

Como você aprendeu, a Biblioteca de Gabaritos Padrões (STL) suporta cinco tipos de iteradores. Dois tipos de iteradores que seus programas podem usar para propósitos específicos são os iteradores de *saída* (para recuperar dados de um repositório e exibir os dados) e os iteradores de *entrada* (para obter dados de uma localização e armazená-los dentro do repositório). Como você aprendeu na Dica 1210, nem os iteradores de *saída* nem os de *entrada* fornecem a funcionalidade total dos iteradores *forward* ou *bidirecionais*. No entanto, os iteradores de *saída* e de *entrada* são úteis dentro do seu código para efetuar suas funções específicas e tornar claro dentro de seu código quais as funções que você quer que os iteradores executem. As limitações nos iteradores de *saída* e de *entrada* são significativas. Por exemplo, seus programas devem usar somente o iterador de *saída* para retornar informações a partir de um determinado repositório, em vez de usá-lo para colocar informações em um determinado repositório, como mostrado aqui:

```
LISTINT::output_iterator sai;

for (sai = listaUm.begin();  sai != listaUm.end(); ++sai)
    cout << *i << " ";

// Este código talvez não funcione corretamente
for (sai = listaUm.begin(); sai != listaUm.end(); ++sai)
    listaUm.insert(1);
```

Como o iterador de saída não atualizará corretamente durante uma operação de inserção, o segundo laço *for* terá efeitos imprevisíveis. Similarmente, seus programas deverão usar iteradores de entrada somente ao efetuar inserções dentro de um repositório e não ao recuperar dados de um repositório, como segue:

```
LISTINT::input_iterator ent;

for (ent = listaUm.begin();  ent != listaUm.end(); ++ent)
    listaUm.insert(1);

// Este código talvez não funcione corretamente
for (ent = listaUm.begin(); ent ! = listaUm end (); ++ent)
    cout << *i << " ";
```

Além de produzir efeitos não-confiáveis após você efetuar uma operação de saída com um iterador *entrada* e após efetuar uma operação de *entrada* com um iterador de *saída*, após incrementar um iterador de *entrada* ou de *saída*, você não pode seguramente comparar, desreferenciar ou incrementar qualquer outra cópia do mesmo iterador.

COMPREENDENDO OS OUTROS TIPOS DE ITERADORES DA BIBLIOTECA DE GABARITOS PADRÕES 1213

Como você aprendeu na Dica 1210 e novamente na Dica 1212, existem limitações significativas em como seus programas podem usar os iteradores de *entrada* e de *saída*. Para evitar as limitações dos iteradores *entrada* e *saída*, seus programas devem usar iteradores *forward*, *bidirecional* ou de *acesso aleatório*. Um iterador *forward X* pode tomar o lugar de um iterador *saída* (para escrever) ou de um iterador *entrada* (para ler). Adicionalmente, você pode ler (usando *V* = **X*) o que você acabou de escrever (usando **X* = *V*) por meio de um iterador *forward*. Além disso, você pode fazer múltiplas cópias de um iterador *forward*, cada uma das quais seu programa poderá desreferenciar e incrementar independentemente.

Além de usar um iterador *forward* para percorrer e acessar os repositórios, seus programas podem usar um iterador *bidirecional*. Os iteradores *bidirecionais* efetuam muito dos mesmos processamentos que os iteradores *forward*. No entanto, você também pode decrementar um iterador *bidirecional*, como em *--X, X--*, ou *(V = *X--)*.

Finalmente, você pode usar um iterador de *acesso aleatório* em vez de um *bidirecional*. Os iteradores *bidirecionais* executam quase o mesmo processamento que os iteradores *bidirecionais* — lhe permitem percorrer a lista da mesma maneira. Além do controle de iteração *bidirecional*, você também pode executar a mesma aritmética de inteiros em um iterador de acesso aleatório que executaria em um objeto ponteiro. Para *N*, um objeto inteiro, você pode escrever *x[N]*, *x + N*, *x - N* e *N + X* para navegar um repositório com inteiros não-iteradores.

Observe que um objeto ponteiro pode tomar o lugar de um iterador de *acesso aleatório* ou aquele de qualquer outro iterador.

Você pode facilmente sumarizar a hierarquia das categorias de iteradores visualizando a hierarquia de iteradores disponíveis para qualquer ação simples. Por exemplo, para acesso de escrita somente em uma seqüência, você pode usar qualquer um dos seguintes iteradores:

```
iterador saída ->
  iterador forward ->
    iterador bidirecional ->
      iterador de acesso aleatório
```

A seta para a direita significa que o iterador à direita e abaixo da seta podem substituir o iterador à esquerda e acima da seta. Portanto, qualquer algoritmo que requer um iterador *saída* deve trabalhar bem com um iterador *forward*, por exemplo. No entanto, você não pode concluir que qualquer algoritmo que requer um iterador *forward* deva trabalhar bem com um iterador *output* (porque o iterador *forward* está à direita e abaixo do operador *saída*).

A hierarquia dos iteradores é similar para acesso de leitura somente a uma seqüência. Para acesso de leitura-somente, seus programas podem usar qualquer um dos seguintes iteradores:

```
iterador entrada ->
  iterador forward ->
    iterador bidirecional ->
      iterador de acesso aleatório
```

Para as ações de leitura somente, um iterador de *entrada* é o mais fraco de todas as categorias de iterador porque ele somente pode percorrer a lista na ordem do início para o fim, e é invalidado por quaisquer atividades de saída. Novamente, qualquer algoritmo que requer um iterador de *entrada* deve trabalhar bem com um iterador *forward* ou qualquer outro iterador abaixo dele na estrutura na forma de árvore. No entanto, você não pode concluir que qualquer algoritmo que requer um iterador *forward* deva trabalhar bem com um iterador *entrada*.

A hierarquia de iteradores é similar para acesso de leitura/somente em uma seqüência. Para acesso de leitura/somente em uma seqüência, seus programas podem usar qualquer um dos seguintes iteradores:

```
iterador forward ->
   iterador bidirecional ->
     iterador de acesso aleatório
```

Lembre-se de que um objeto ponteiro sempre pode servir como um iterador de *acesso aleatório*. Portanto, ele pode servir como qualquer categoria de iterador, já que suporta o acesso de leitura/escrita correto à seqüência que designa. Esse iterador "álgebra" é fundamental para quase todos os outros processamentos que seus programas executarão com os algoritmos e repositórios da Biblioteca de Gabaritos Padrães. É importante compreender as capacidades e limitações da categoria de cada iterador para ver como os repositórios e os algoritmos na Biblioteca de Gabaritos Padrão usam iteradores.

1214 COMPREENDENDO OS CONCEITOS

Como você aprendeu, é importante analisar suas classes genéricas para determinar se é possível chamar corretamente esses repositórios de classes genéricas, pois você pode efetuar certas atividades com uma classe repositório que não pode efetuar de forma tão eficiente como uma classe de objeto simples. No entanto, é igualmente importante determinar outras informações sobre a classe ou função genérica. Uma pergunta importante a fazer sobre qualquer função de gabarito, não apenas sobre os algoritmos da Biblioteca de Gabaritos Padrões, é qual o tipo *set* que seus programas podem substituir corretamente para os parâmetros de gabaritos padrões. Para compreender melhor a importância da substituição correta, considere o algoritmo *find*, definido no arquivo de cabeçalho *algo.h* da Biblioteca de Gabaritos Padrões, como mostrado aqui:

```
InputIterator find (InputIterator first, InputIterator last, const T& value)
 {
   while (first != last && *first != value)
      ++first;
   return first;
 };
```

Quando você olhar atentamente a definição para o algoritmo *find*, verá que ela usa a aritmética de incremento padrão para negociar os repositórios. Por exemplo, isso significa que você pode substituir um ponteiro do tipo *int** ou do tipo *double** pelo parâmetro formal do gabarito *InputIterator* de *find*. No entanto, você não pode substituir *int* ou *double*, pois *find* usa a expressão **first*, e o operador de desreferência não faz sentido para um objeto do tipo *int* ou do tipo *double*. Fundamentalmente, então, *find* define de forma implícita um conjunto de requisitos nos tipos. Portanto, você pode usar *find* com qualquer tipo que satisfaça esses requisitos. Basicamente, qualquer tipo que você substituir para *InputIterator* precisa fornecer certas operações: o tipo precisa ser capaz de comparar dois objetos daquele tipo para ver se são iguais, incrementar um objeto daquele tipo, desreferenciar um objeto daquele tipo, obter o objeto para o qual ele aponta e assim por diante.

Find não é o único algoritmo da Biblioteca de Gabaritos Padrões que tem esse conjunto de requisitos; os argumentos para *for_each* e *count*, por exemplo, bem como muitos outros algoritmos da Biblioteca de Gabaritos Padrões, precisam satisfazer os mesmos requisitos. Os desenvolvedores da Biblioteca de Gabaritos Padrões chamam esse importante conjunto de requisitos de tipo de *conceito*. O conceito particular do caso da função *find* é o conceito *Input Iterador*.

1215 COMPREENDENDO OS MODELOS

Na dica anterior vimos que, um conceito define um conjunto de requisitos a que uma função de gabarito precisa atender. Em geral, um tipo atende a um conceito (isto é, um tipo é um *modelo* de um conceito) se o tipo satisfaz todos os requisitos do conceito Portanto, você pode corretamente dizer que *int** é um modelo do conceito Iterador de Entrada porque *int** fornece todas as operações que os requisitos do Iterador da Entrada especifica.

Os conceitos não são parte da linguagem C++. Em outras palavras, você não pode declarar um conceito em um programa ou declarar que um tipo em particular seja um modelo de um conceito. No entanto, os conceitos e seus modelos correspondentes são componentes da Biblioteca de Gabaritos Padrões extremamente importantes. Usar componentes torna possível escrever programas que separam a interface da implementação. Por exemplo, o autor do algoritmo *find* apenas precisa considerar a interface que o conceito Iterador de Entrada especifica, em vez de a implementação de todo tipo possível que adere a esse conceito. Similarmente, se você quiser usar *find*, apenas precisa garantir que os argumentos que você passa para ele são modelos do conceito Iteradores de Entrada (tal como ponteiros de entrada).

A separação da interface da implementação (que é o objeto principal de todas as funções e classes genéricas, não apenas das funções e classes da Biblioteca de Gabaritos Padrões) é a razão pela qual você pode usar *find* e *reverse* com *listas*, *vetores*, matrizes C e muitos outros tipos de repositório. Basicamente, programar em termos de conceitos, em vez de em termos de tipos específicos, lhe permite reutilizar software e combinar componentes de software.

COMPREENDENDO OS ALGORITMOS 1216

Como você aprendeu, o repositório é um bloco de construção fundamental da Biblioteca de Gabaritos Padrões. A Biblioteca de Gabaritos Padrões também inclui uma grande coleção de algoritmos que seus programas podem usar para manipular a data que você armazena nos repositórios. Por exemplo, você pode usar o algoritmo *reverse* para inverter a ordem do elemento em um vetor, como mostrado aqui:

```
reverse(v.begin(), v.end());
```

Existem dois pontos importantes a observar sobre essa chamada à função *reverse*. Primeiro, *reverse* é uma função global, não uma função-membro. Segundo, *reverse* utiliza dois argumentos em vez de um — o que significa que *reverse* opera em uma gama de elementos dentro de um repositório, em vez de no próprio repositório. Neste exemplo em particular, o intervalo é todo o repositório *v* (pois os parâmetros são *v.begin* e *v.end*).

A razão pela qual *reverse* é uma função global e atual nos elementos dentro de um repositório, em vez de no próprio repositório, é simples. Como outros algoritmos da Biblioteca de Gabaritos Padrões, *reverse* está "desacoplada" das classes de repositório da Biblioteca de Gabaritos Padrões. O desacoplamento das classes individuais significa não somente que você pode usar *reverse* para inverter os elementos nos vetores, mas também inverter os elementos nas listas e até os elementos nas matrizes C. Por causa do desacoplamento do algoritmo *reverse*, até mesmo o programa a seguir, *rever_mz.cpp*, é válido:

```
#include <iostream.h>
#include <algorith.h>

using namespace std ;

void main(void)
 {
   double A[6] = { 1.2, 1.3, 1.4, 1.5, 1.6, 1.7 };
   int i;
   reverse(A, A + 6);
   for (i = 0; i < 6; ++i)
     cout << "A[" << i << "] = " << A[i] << "  ";
   cout << endl;
 }
```

Esse exemplo usa um intervalo, exatamente como fez o exemplo da inversão de um vetor. Quando este livro se refere a um intervalo dessa natureza (isto é, um intervalo cujo segundo argumento aponta para além do final do intervalo), ele denotará o intervalo como [início,fim). O intervalo usa a notação assimétrica (isto é, o colchete na esquerda e o parênteses na direita) como um lembrete que os dois pontos finais são diferentes. O primeiro ponto final é o início do intervalo, mas o segundo ponto final está um além do final do intervalo. Quando você compilar e executar o programa *rever_mt.cpp*, sua tela exibirá a seguinte saída:

```
A[0] = 1.2  A[1] = 1.3  A[2] = 1.4  A[3] = 1.5  A[4] = 1.6  A[5] = 1.7
A[0] = 1.7  A[1] = 1.6  A[2] = 1.5  A[3] = 1.4  A[4] = 1.3  A[5] = 1.2
C:\
```

USANDO OUTRO EXEMPLO DE ALGORITMO DA BIBLIOTECA DE GABARITOS PADRÕES 1217

Na dica anterior, você aprendeu sobre os algoritmos da Biblioteca de Gabaritos Padrões (STL) e como seus programas podem usá-los, tanto com os repositórios da Biblioteca de Gabaritos Padrões quanto com outros tipos de repositórios. Na Dica 1214, você aprendeu sobre o algoritmo *find* da Biblioteca de Gabaritos Padrões. Seus programas podem usar o algoritmo *find* para localizar ocorrências de um objeto contido dentro de um repositório. Você geralmente usará o algoritmo *find* para localizar um nó individual, por exemplo, dentro de um repositório da classe *list*. Adicionalmente, você também pode usar o algoritmo *find* para localizar elementos individuais dentro de uma matriz, como mostra o programa *mtz_loc.cpp* a seguir:

```
#include <iostream.h>
```

```
#include <algorith.h>

using namespace std ;

void main(void)
 {
   const int MATRIZ_TAM = 8;
   int IntMatriz[MATRIZ_TAM] = { 1, 2, 3, 4, 4, 5, 6, 7 };
   int *posicao;  // armazena a posição do primeiro elemento igual.
   int i;
   int valor = 4;

   // imprime o conteúdo de IntMatriz
   cout << "IntMatriz { ";
   for (i = 0; i < MATRIZ_TAM; i++)
       cout << IntMatriz[i] << ", ";
   cout << " \b }" << endl;

   // Localiza o primeiro elemento no intervalo
   // [primeiro, último + 1) que seja igual ao valor.
   posicao = find(IntMatriz, IntMatriz + MATRIZ_TAM, valor);

   // imprime o elemento igual se um foi encontrado
   if (posicao != IntMatriz + MATRIZ_TAM) // elemento igual encontrado
     cout << "Primeiro elemento que é igual a " << valor << " está na posição "
          << (posicao - IntMatriz) << endl;
   else       // nenhum elemento igual encontrado
     cout << "A seqüência não contém qualsquer "
          << " elementos com valor " << valor << endl;
 }
```

O programa *mtz_loc.cpp* cria primeiro uma matriz de valores e depois pesquisa um item dentro dessa matriz. O programa usa o algoritmo *find* para determinar se o item está dentro da matriz. Quando você compilar e executar o programa *mtz_loc.cpp*, sua tela exibirá a seguinte saída:

```
IntMatriz { 1, 2, 3, 4, 4, 5, 6, 7 }
O primeiro elemento igual a 4 está na posição 3
C:\>
```

1218 DESCREVENDO OS ALGORITMOS QUE A STL INCLUI

A Biblioteca de Gabaritos Padrões (STL) fornece um número grande de algoritmos que seus programas podem usar quando manipulam tanto os objetos da Biblioteca de Gabaritos Padrões quanto objetos que não são da Biblioteca de Gabaritos Padrões. A Biblioteca de Gabaritos Padrões divide seus algoritmos em quatro tipos básicos de algoritmos. O primeiro tipo é o conjunto de *algoritmos não-mutantes*. A Tabela 1218.1 lista os algoritmos não-mutantes da Biblioteca de Gabaritos Padrões.

Tabela 1218.1 Algoritmos não-mutantes da Biblioteca de Gabaritos Padrões.

Algoritmos Não-Mutantes			
for_each	*find*	*find_if*	*adjacent_find*
find_first_of	*count*	*count_if*	*mismatch*
equal	*search*	*search_n*	*find_end*

O segundo tipo de algoritmo da Biblioteca de Gabaritos Padrões é o conjunto de *algoritmos mutantes*. Como regra, os algoritmos mutantes mudam a natureza dos objetos dentro de um repositório ou copia esses objetos para outros repositórios. A Tabela 1218.2 lista os algoritmos mutantes da Biblioteca de Gabaritos Padrões.

Tabela 1218.2 Os algoritmos mutantes da Biblioteca de Gabaritos Padrões.

Algoritmos Mutantes			
copy	*copy_n*	*copy_backward*	*swap*
iter_swap	*swap_ranges*	*transform*	*replace*
replace_if	*replace_copy*	*replace_copy_if*	*fill*
fill_n	*generate*	*generate_n*	*remove*
remove_if	*remove_copy*	*remove_copy_if*	*unique*
unique_copy	*reverse*	*reverse_copy*	*rotate*
rotate_copy	*random_shuffle*	*random_sample*	*random_sample_n*
partition	*stable_partition*		

O terceiro tipo de algoritmo da Biblioteca de Gabaritos Padrões é o conjunto de *algoritmos de classificação.* Embora os algoritmos de classificação sejam tecnicamente um subconjunto dos algoritmos mutáveis, o conjunto de algoritmos de classificação é bem extenso. A Tabela 1218.3 lista os algoritmos de classificação da Biblioteca de Gabaritos Padrões.

Tabela 1218.3 Os algoritmos de classificação da Biblioteca de Gabaritos Padrões.

Algoritmos de Classificação			
sort	*stable_sort*	*partial_sort*	*partial_sort_copy*
is_sorted	*nth_element*	*lower_bound*	*upper_bound*
equal_range	*binary_search*	*merge*	*inplace_merge*
includes	*set_union*	*set_intersection*	*set_difference*
set_symmetric_difference	*push_heap*	*pop_heap*	*make_heap*
sort_heap	*is_heap*	*min*	*max*
min_element	*max_element*	*lexicographical_compare*	
lexicographical_compare_3way		*next_permutation*	*prev_permutation*

Finalmente, a Biblioteca de Gabaritos Padrões também suporta um conjunto de algoritmos numéricos generalizados. Seus programas podem usar os algoritmos numéricos generalizados para efetuar atividades matemáticas em um tipo desconhecido de número (por exemplo, um *float* e um *double* ou um *float* e um *int*), dependendo do algoritmo específico e seu propósito. A Tabela 1218.4 lista os algoritmos numéricos generalizados da Biblioteca de Gabaritos Padrões.

Tabela 1218.4 Os algoritmos numéricos generalizados da Biblioteca de Gabaritos Padrões.

Algoritmos Numéricos Generalizados		
iota	*accumulate*	*inner_product*
partial_sum	*adjacent_difference*	*power*

Claramente, a lista de algoritmos que a Biblioteca de Gabaritos Padrões define é muito extensa. Infelizmente, este livro não abordará todos os algoritmos. No entanto, você pode consultar a documentação do seu compilador ou a documentação da Biblioteca de Gabaritos Padrões para obter maiores informações sobre qualquer algoritmo listado nesta dica.

ESTUDANDO O ALGORITMO FOR_EACH DA STL 1219

Como você aprendeu, a Biblioteca de Gabaritos Padrões (STL) define diferentes tipos de algoritmos. Embora você já tenha usado o algoritmo não-mutável *find*, mesmo assim vale a pena examinar um segundo programa de exemplo, que usa o algoritmo *for_each* não-mutável. O algoritmo *for_each* chama alguma função *Func1* para cada elemento no intervalo (primeiro, último) e não retorna nenhum valor, como mostrado aqui:

```
void for_each(primeiro, ultimo, func1);
```

O algoritmo *for_each* não modifica quaisquer elementos na seqüência. Por exemplo, o programa a seguir, *terc_pot.cpp*, usa o algoritmo *for_each* para acessar cada elemento em um vetor e exibir o cubo desse elemento:

```
#include <iostream>
#include <vector>
#include <algorith.h>
using namespace std;

void ImprimeCubo(int n)
 {
    // imprime o cubo do inteiro n
    cout << "O cubo de " << n << " é " << n * n * n << endl;
 }

void main(void)
 {
    const int TAM_VETOR = 8 ;
    typedef vector<int > IntVetor ;        // Define um vetor de inteiros
    typedef IntVetor::iterator IntVetorIt;       // Define um tipo iterador
    IntVetor Numbers(TAM_VETOR);        // vetor que contém números
    IntVetorIt inicio, fim, it; // iteradores
    int i ;

    for (i = 0; i < TAM_VETOR; i++)     // Inicializa o vetor Numeros
        Numeros[i] = i + 1 ;

    inicio = Numeros.begin() ;        // posição do primeiro elemento de
     Numeros
    fim = Numeros.fim() ;        // um além da posição do último elemento de
       // Numeros

    cout << "Numeros { " ;         // imprime o conteúdo de Numeros
    for(it = inicio; it != fim; it++)
        cout << *it << " " ;
    cout << " }\n" <<endl ;

    // para cada elemento no intervalo
(primeiro, ultimo), imprime o cubo dos elementos
    for_each(inicio, fim, ImprimeCubo);
 }
```

O programa *terc_pot.cpp* inicialmente cria o vetor *IntVetor*, depois atribui um valor a cada objeto no vetor. Após exibir os valores do vetor, o programa usa o algoritmo *for_each* para exibir o cubo do elemento de cada vetor. Quando você compilar e executar o programa *terc_pot.cpp*, sua tela exibirá a seguinte saída:

```
Numeros  { 1 2 3 4 5 6 7 8 }

O cubo de 1 é 1
O cubo de 2 é 8
O cubo de 3 é 27
O cubo de 4 é 64
O cubo de 5 é 125
O cubo de 6 é 216
O cubo de 7 é 343
O cubo de 8 é 512
C:\>
```

ESTUDANDO O ALGORITMO GENERATE_N DA STL 1220

Como você aprendeu na Dica 1218, a Biblioteca de Gabaritos Padrões (STL) suporta um grande número de algoritmos que seus programas podem usar para manipular repositórios e outros objetos. Um dos algoritmos mutantes mais exclusivos que a Biblioteca de Gabaritos Padrões suporta é *generate_n*, que preenche cada objeto dentro de um intervalo de objetos dentro de um repositório com o valor de retorno de uma função geradora. A função geradora retorna um valor que o algoritmo coloca naquele objeto. Para compreender melhor como a função geradora retorna o valor, considere o seguinte programa, *gera_fib.cpp*, que usa o algoritmo *generate_n* para colocar números da seqüência de Fibonacci dentro de um vetor:

```
//  Se você usa o Visual C++, defina as opções do compilador
//  como /GX
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// retorna o próximo número na série de Fibonacci.
int Fibonacci(void)
 {
    static int r;
    static int f1 = 0;
    static int f2 = 1;
    r = f1 + f2;
    f1 = f2;
    f2 = r;
    return f1;
 }

void main(void)
 {
    const int TAM_VETOR = 15;

    // Define uma classe gabarito de vetor de inteiros
    typedef vector<int> IntVetor;

    // Define um iterador para a classe de gabarito de
    // vetor de inteiros
    typedef IntVetor::iterator IntVectorIt;

    IntVetor Numeros(TAM_VETOR);          // vetor que contém números
    IntVectorIt inicio, fim , it;
    int i;

    //Inicializa o vetor Numeros
    for(i = 0; i < TAM_VETOR; i++)
        Numeros[i] = i * i;
    inicio = Numeros.begin();        // posição do primeiro elemento de Numeros
    fim = Numeros.fim();         // um além da posição do último elemento
       // de Números
    cout << "Antes de chamar generate_n" << endl;
      // imprime o conteúdo de Números
    cout << "Números { ";
    for (it = inicio; it != fim; it++)
        cout << *it << " ";
    cout << " }\n" << endl;

    // preenche o intervalo especificado com uma série
    // de números de Fibonacci usando a função Fibonacci
```

```
    generate_n(inicio + 5, Numeros.size() - 5, Fibonacci);

    cout << 'Após chamar generate_n" << endl;

    // imprime o conteúdo de Numeros
    cout << 'Numeros (";
    for (it = inicio; it != fim; it++)
    cout << *it << " ";
    cout << ' )\n" << endl;
 }
```

O programa *gen_fib.cpp* inicializa o vetor *numeros* para conter 14 valores, cada um dos quais contém o quadrado do índice dentro do vetor. Por exemplo, o valor no índice 1 contém 1 (1 * 1) e o valor no índice 12 contém 144 (12 * 12). Em seguida, o programa substitui números dentro do vetor começando no sexto número, que ele substitui dentro do primeiro número na seqüência de Fibonacci. O programa substitui cada número restante dentro do vetor com o próximo número na seqüência de Fibonacci. Quando você compilar e executar o programa *gen_fibo.cpp*, sua tela exibirá a seguinte saída:

```
Antes de chamar generate_n
Números  ( 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 )

Após chamar generate_n
Números  ( 0 1 4 9 16 1 1 2 3 5 8 13 21 34 55 )
C:\>
```

1221 COMPREENDENDO O ALGORITMO RANDOM_SHUFFLE DA STL

À medida que seus programas se tornarem mais complexos, algumas vezes você precisará trocar as posições das informações dentro de um repositório em alguma ordem aleatória. Geralmente, você efetuará tal processamento quando escrever jogos ou outros programas que simulam respostas inteligentes do computador. Para simplificar tal processamento, a Biblioteca de Gabaritos Padrões (STL) fornece o algoritmo *random_shuffle*. O algoritmo *random_shuffle* permuta os elementos de uma seqüência (do primeiro até o último) em uma ordem aleatória. O algoritmo usa um gerador de números aleatórios internos para gerar os índices dos elementos a permutar ou alimentar o gerador de números aleatórios com o número de elementos que o repositório inclui, dependendo de sua invocação. Tanto a versão predicado quanto a não-predicado de *random_shuffle* usam o operador = para efetuar as permutas. Para compreender melhor o processamento que o algoritmo *random_shuffle* executa, considere o programa *shuffle.cpp*, como mostrado aqui:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>

using namespace std;

void main(void)
 {
    const int TAM_VETOR = 8 ;
    typedef vector<int> IntVetor;       // Define um gabarito da classe
    typedef IntVetor::iterator IntVetorIt ;     // Define um iterador para
      gabarito da classe vetor
    IntVetor Numeros(TAM_VETOR) ;

    IntVetorIt inicio, fim, it ;     // Inicializa o vetor Numeros

    Numeros[0] = 4 ;
    Numeros[1] = 10;
```

```
    Numeros[2] = 70;
    Numeros[3] = 30;
    Numeros[4] = 10;
    Numeros[5] = 69;
    Numeros[6] = 96;
    Numeros[7] = 100;

    inicio = Numeros.begin() ; // posição do primeiro elemento de Numeros
    fim = Numeros.fim();       // um além da posição do último elemento de
    // Numeros

    cout << "Antes de chamar random_shuffle\n" << endl;

    cout << "Números ( " ;       // imprime o conteúdo de Numeros
    for(it = inicio; it != fim; it++)
        cout << *it << " " ;
    cout << "\b )\n" <<endl ;

    random_shuffle(inicio, fim) ;    // embaralha os elementos em uma ordem
    // aleatória
    cout << "Após chamar random_shuffle\n" << endl;
    cout << "Números ( " ;
    for(it = inicio; it != fim; it++)
        cout << *it << " " ;
    cout << "\b )\n" << endl ;
}
```

Primeiro, o programa *shuffle.cpp* cria um vetor e o preenche com uma série de números. Segundo, o programa chama o algoritmo *random_shuffle* e o instrui a permutar todos os números no vetor. Finalmente, o programa exibe os números novamente após completar o *random_shuffle*. Quando você compilar e executar o programa *shuffle.cpp*, sua tela exibirá saída similar à seguinte:

```
Antes de chamar random_shuffle
Números  ( 4 10 70 30 10 69 96 100 )

Após chamar random_shuffle
Números  ( 96 4 69 70 10 30 100 )
C:\>
```

USANDO O ALGORITMO PARTIAL_SORT_COPY 1222

Como você aprendeu na Dica 1218, a Biblioteca de Gabaritos Padrões suporta muitos algoritmos com diferentes funções. Em dicas anteriores, você escreveu pequenos programas que usam algoritmos mutantes e não-mutantes da Biblioteca de Gabaritos Padrões. Na dica a seguir, você usará o algoritmo *merge* para combinar os valores em dois vetores. Como você aprendeu, a função *merge* é um dos algoritmos de classificação da Biblioteca de Gabaritos Padrões. Nesta dica, você usará outro algoritmo de classificação, o *partial_sort_copy*.

O algoritmo *partial_sort_copy* classifica os N menores elementos, onde $N = min((último1 - primeiro1), (último2 - primeiro2))$, da seqüência *[primeiro1, último1)* e copia os resultados para a seqüência *[primeiro2, primeiro2 + N)*. Em outras palavras, o algoritmo *partial_sort_copy* classifica o repositório em um repositório temporário na ordem ascendente. O algoritmo *partial_sort_copy* então determina o menor número dentro do repositório temporário (o primeiro número). Ele depois copia esse número de objeto do repositório temporário classificado para um terceiro repositório, permanente. Por exemplo, se uma matriz contiver os valores 15, 25, 7, 8, 10, 12 e 2, a matriz inicialmente aparecerá dentro da memória, como mostrado na Figura 1221.1.

Figura 1222.1 A matriz antes da classificação ou da cópia.

O algoritmo *partial_sort_copy* então classificará a matriz e a armazenará dentro de uma matriz temporária, como mostrado na Figura 1222.2.

Figura 1222.2 A matriz temporária classificada.

A função *partial_sort_copy* então determinará o menor número no resultado classificado (neste caso, 2) e copiará os dois primeiros valores da matriz temporária para a cópia matriz permanente, deixando duas matrizes, como mostrado na Figura 1222.3.

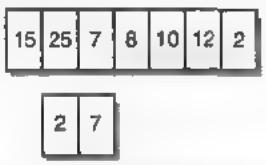

Figura 1222.3 As duas matrizes que resultam da chamada de *partial_sort_copy*.

Para compreender melhor o processamento que o algoritmo *partial_sort_copy* executa, considere o programa a seguir, *ps_copia.cpp*, que executa os passos descritos nas figuras anteriores:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>

using namespace std;

void main(void)
 {
    const int TAM_VETOR = 8;

    // Define um gabarito de classe vetor de int
    typedef vector<int> IntVetor;

    // Define um iterador para gabarito de classe vetor de strings
    typedef IntVetor::iterator IntVetorIt;

    IntVetor Numeros(TAM_VETOR);
    IntVetor Result(4);
    IntVetorIt inicio, fim, it;

    // Inicializa o vetor Numeros
    Numeros[0] = 4;
    Numeros[1] = 10;
    Numeros[2] = 70;
    Numeros[3] = 30;
    Numeros[4] = 10;
    Numeros[5] = 69;
    Numeros[6] = 96;
    Numeros[7] = 7;

    inicio = Numeros.begin();        // posição do primeiro elemento de Numeros
    fim = Numeros.fim();          // um além da posição do último elemento de
        // Numeros
    cout << "Antes de chamar partial_sort_copy\n"          << endl;

                                          // imprime o conteúdo de Numeros
    cout << "Números { ";
    for (it = inicio; it != fim; it++)
        cout << *it << " ";
    cout << "\b }\n" << endl;
```

```
    // classifica os 4 menores elementos em Numeros e
    // copia os resultados em Result
    partial_sort_copy(inicio, fim, Result.begin(), Result.fim());

    cout << "Após chamar partial_sort_copy\n" << endl;
    cout << "Números { ";
    for(it = inicio; it != fim; it++)
        cout << *it << " ";
    cout << "\b }\n" << endl;

    cout << "Result { ";
    for(it = Result.begin(); it != Result.fim(); it++)
        cout << *it << " ";
    cout << "\b }\n" << endl;
}
```

Quando você compilar e executar o programa *ps_copy.cpp*, ele criará um vetor de oito elementos inteiros. Ele então atribuirá valores a esses elementos. Para mostrar esse processamento, *ps_copia.cpp* exibe os elementos do vetor antes de chamar *partial_sort_copy*. Dentro de *partial_sort_copy*, o algoritmo determina que 4 é o menor número e copia os primeiros quatro valores na matriz classificada para a matriz de saída. Quando você compilar e executar o programa *ps_copia.cpp*, sua tela exibirá a seguinte saída:

```
Antes de chamar partial_sort_copy
Números  { 4 10 70 30 10 69 96 7  }

Após chamar partial_sort_copy
Números  { 4 10 70 30 10 69 96 7  }
Resultado  { 4 7 10 10  }
C:\>
```

COMPREENDENDO O ALGORITMO MERGE 1223

Na dica anterior, você usou o algoritmo *partial_sort_copy* para criar uma cópia parcialmente classificada de um vetor. Nesta dica, você usará o algoritmo *merge* para mesclar dois vetores classificados em um terceiro vetor classificado que conterá todos os valores de ambos os vetores originais. Você usará o algoritmo *merge* dentro de seus programas, como mostrado no protótipo a seguir:

```
merge(primeiro1, ultimo1, primeiro2, ultimo2, resultado);
```

O algoritmo *merge* mescla duas seqüências classificadas (*primeiro1...ultimo1*) e (*primeiro2...ultimo2*) em uma única seqüência iniciando no iterador *resultado*. O algoritmo *merge* assume que os intervalos (*primeiro1...ultimo1*) e (*primeiro2...ultimo2*) foram classificados anteriormente usando o operador menor que. Se ambos os intervalos contêm valores iguais, *merge* primeiro armazena o valor do primeiro intervalo dentro do intervalo resultante. Para compreender melhor o processamento do algoritmo *merge*, considere o seguinte programa, *merge_2v.cpp*:

```
#include <iostream.h>
#include <vector.h>
#include <algorith.h>
#include <list.h>
#include <deque.h>

using namespace std;

void main(void)
 {
    const int MAX_ELEMENTOS = 8;
    typedef vector<int> IntVetor;       // Define um gabarito de classe
     // vetor de int
    typedef IntVetor::iterator IntVetorIt;        // Define um tipo iterator
    typedef list<int> IntList;       // Define um gabarito de classe lista
```

```
// de int
typedef IntList::iterator IntListIt;        // Define um tipo iterador
typedef deque<int> IntDeque;        // Define um gabarito de classe
// deque de int
typedef IntDeque::iterator IntDequeIt;       // Define um tipo iterador

IntVetor NumerosVetor(MAX_ELEMENTOS);
IntVetorIt iniciov, fimv, itv;
IntList NumerosLista;
IntListIt primeiro, ultimo, itl;
IntDeque NumerosDeque(2 * MAX_ELEMENTOS);
IntDequeIt itd;

// Inicializa o vetor NumerosVetor
NumerosVetor[0] = 4;
NumerosVetor[1] = 10;
NumerosVetor[2] = 70;
NumerosVetor[3] = 10;
NumerosVetor[4] = 30;
NumerosVetor[5] = 69;
NumerosVetor[6] = 96;
NumerosVetor[7] = 100;

iniciov = NumerosVetor.begin();       // posição do primeiro elemento de
// NumerosVetor
fimv = NumerosVetor.end();       // um além do último elemento de
// NumerosVetor

// classifica NumerosVetor, merge requer que as
// seqüências estejam classificadas
sort(iniciov, fimv);

// imprime o conteúdo de NumerosVetor
cout << "NumerosVetor { ";
for(itv = iniciov; itv != fimv; itv++)
    cout << *itv << " ";
cout << " }\n" << endl;

// Inicializa a lista NumerosLista
for(int i = 0; i < MAX_ELEMENTOS; i++)
NumerosLista.push_back(i);
primeiro = NumerosLista.begin();     // posição do primeiro elemento do
// NumerosLista
ultimo = NumerosLista.end();     // um além do último elemento de
// NumerosLista

cout << "NumerosLista { ";     // imprime o conteúdo de NumerosLista
for(itl = primeiro; itl != ultimo; itl++)
    cout << *itl << " ";
cout << " }\n" << endl;

// mescla os elementos de NumerosVetor e NumerosLista
// e coloca os resultados em NumerosDeque
merge(iniciov, fimv, primeiro, ultimo, NumerosDeque.begin());

// imprime o conteúdo de NumerosDeque
cout << "Após chamar merge\n" << endl;
cout << "NúmerosDeque { " ;
for (itd = NumerosDeque.begin();
     itd != NumerosDeque.end(); itd++)
   cout << *itd << " ";
```

```
    cout << " }\n" << endl;
}
```

O programa *merge_2v.cpp* primeiro cria e inicializa o vetor *NumerosVetor*. Em seguida, ele chama a função *sort* para colocar o vetor *numeros* em ordem ascendente. O programa então cria a lista *NumerosLista*, que o código cria totalmente classificada. Após *merge_2v.cpp* criar a lista *NumerosLista*, ele chama a função *merge* e coloca o resultado dentro do repositório *NumerosDeque*. Para mostrar que completou seu processamento, *merge_2v.cpp* exibe os números que o repositório *NumerosDeque* contém. Quando você compilar e executar o programa *merge_2v.cpp*, sua tela exibirá o seguinte resultado:

```
NumerosVetor  { 4 10 10 30 69 70 96 100 }
NumerosLista  { 0 1 2 3 4 5 6 7 }
Após chamar merge
NumerosDeque { 0 1 2 3 4 4 5 6 7 10 10 30 69 70 96 100 }
= C:\>
```

COMPREENDENDO O ALGORITMO INNER_PRODUCT 1224

Na dica anterior, você usou o algoritmo *merge* para mesclar um *vetor* classificado e uma *lista* classificada em um *deque* classificado que contém todos os valores de ambos os objetos originais. Na verdade, em dicas recentes você usou os algoritmos da Biblioteca de Gabaritos Padrões de três ou quatro tipos principais. mutantes, não-mutantes e classificação. Nesta dica, você usará um dos algoritmos numéricos generalizados, o algoritmo *inner_product*. Esse algoritmo multiplica *N* elementos dentro de dois repositórios um pelo outro e retorna a soma das multiplicações. Por exemplo, se você calcular o produto interno de duas matrizes começando no segundo elemento e continuando por três elementos, o valor *inner_product* será consistente com o mostrado na Figura 1224.1

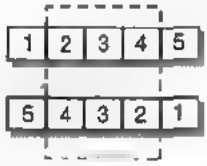

Figura 1224.1 Como o algoritmo *inner_product* calcula o resultado do produto interno da "soma dos produtos" de duas matrizes.

O algoritmo *inner_product* também suporta uma segunda implementação sobrecarregada, que é o produto das somas dos valores internos. Usando o exemplo mostrado na Figura 1224.1, você determinará o produto do produto interno da soma, como mostrado na Figura 1224.2.

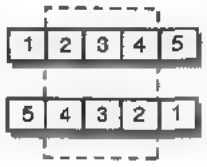

Figura 1224.2 Como o algoritmo *inner_product* calcula o resultado do produto interno do "produto da soma" de duas matrizes.

Para compreender melhor o processamento que o algoritmo *inner_product* executa, considere o seguinte programa, *in_prod.cpp*:

```
// Se você usar o Visual C++, defina as opções do
```

```
//  compilador com /GX
#include <iostream.>
#include <vector.>
#include <numeric.>
#include <iterator.>

using namespace std;

typedef vector<float> MatrizFloat;
typedef ostream_iterator<float, char, char_traits<char>>
FloatOstreamIt;

void main(void)
 {
    FloatOstreamIt itOstream(count, " ");

    // Inicializa as matrizes
    MatrizFloat rgF1, rgF2;
    for (int i=1; i<=5; i++)
    {
        rgF1.push_back(i);
        rgF2.push_back(i*i);
    };

    // imprime as matrizes
    cout << "Matriz 1: ";
    copy(rgF1.begin(), rgF1.end(), itOstream);
    cout << endl;
    cout << "Matriz 2: ";
    copy(rgF2.begin(), rgF2.end(), itOstream);
    cout << endl;

    // Esta é a soma dos produtos (S.O.P) dos elementos correspondentes
    float ip1 = inner_product(rgF1.begin(), rgF1.end(), rgF2.begin(), 0);
    cout << "O produto interno (S.O.P.) de Matriz1 e "
         << "Matriz2 é " << ip1 << endl;

    // Este é o produto das somas (P.O.S.) dos elementos correspondentes
    float ip2 = inner_product(rgF1.begin(), rgF1.end(), rgF2.begin(), 1,
          multiplies<float>(), plus<float>());
    cout << "O produto interno (P.O.S.) de Matriz2 é "
         << ip2 << endl;
 }
```

O programa *in_prod.cpp* começa inicializando dois vetores do tipo *float* e preenchendo os vetores com valores — o primeiro, com uma seqüência direta; e o segundo, com o quadrado da seqüência direta. O programa exibe os dois vetores para sua revisão. Em seguida, o programa calcula a soma do produto *inner_product* (S.O.P) e completa calculando o produto da soma de *inner_product* (P.O.S.). Quando você compilar e executar o programa *in_prod.cpp*, sua tela exibirá a seguinte saída:

```
Matriz 1: 1 2 3 4 5
Matriz 2: 1 4 9 16 25
O produto interno (S. D. P.) de matriz1 e matriz2 é 225
O produto interno (P. D. S.) de matriz1 e matriz2 é 86400
C:\>
```

COMPREENDENDO MELHOR OS VETORES 1225

Como você aprendeu, um *vetor* é um Repositório de Seqüência que suporta o acesso aleatório aos elementos, inserção e remoção de elementos em tempo constante no final do repositório (isto é, a quantidade de tempo que o programa usa para inserir elementos não varia com base no tamanho do *vetor*) e a inserção e remoção de elementos em tempo linear no início ou no meio do repositório (a quantidade de tempo que o programa usa para inserir elementos varia dependendo do tamanho do *vetor*). O número de elementos em um vetor pode variar dinamicamente. Como você aprenderá, como o vetor pode realocar mais memória para si mesmo, seu gerenciamento de memória é automático. A classe *vector* é a mais simples das classes repositório da Biblioteca de Gabaritos Padrões e, em muitos casos, a mais eficiente. Como você aprendeu, um vetor é similar a uma matriz. Por exemplo, o fragmento de código a seguir mostra como você declara e usa um vetor:

```
vector<int> V;
V.insert(V.begin(), 3);
```

A primeira linha declara um *int vector V*, que não contém elementos. A segunda linha insere o valor 3 no início do vetor. Além das funções-membro *insert* e *begin*, a classe *vector* suporta muitas funções-membro. A Tabela 1225 lista as funções-membro para a classe *vector* e uma breve descrição de cada uma.

Tabela 1225 Os membros da classe *vector*.

Membro	Descrição
value_type	O tipo do objeto, *T*, armazenado no vetor.
pointer	Ponteiro para *T*.
reference	Referência para *T*.
const_reference	Referência *const* para *T*.
size_type	Um tipo inteiro sem sinal.
difference_type	Um tipo inteiro com sinal.
iterator	A definição do tipo iterador base que seu programa deve usar para iterar dentro do vetor.
const_iterator	A definição do tipo iterador base *Const* que seu programa deve usar para iterar dentro de um vetor
reverse_iterator	A definição do tipo iterador base que seu programa deve usar para iterar de trás para frente dentro de um vetor.
const_reverse_iterator	A definição do tipo iterador base *Const* que seu programa deve usar para iterar de trás para a frente dentro de um vetor.
iterator begin()	Retorna um iterador que aponta para o início do vetor.
iterator end()	Retorna um iterador que aponta para o final do vetor.
const_iterator begin() const	Retorna um *const_iterator* que aponta para o início do vetor.
const_iterator end() const	Retorna um *const_iterator* que aponta para o final do vetor.
reverse_iterator rbegin()	Retorna um *reverse_iterator* que aponta para o início do vetor invertido.
reverse_iterator rend()	Retorna um *reverse_iterator* que aponta para o final do vetor invertido.
const_reverse_iterator rbegin() const	Retorna um *const_reverse_iterator* que aponta para o início do vetor invertido.
const_reverse_iterator rend() const	Retorna um *const_reverse_iterator* que aponta para o final do vetor invertido.
size_type size() const	Retorna o tamanho do vetor em elementos.
size_type max_size() const	Retorna o maior tamanho possível para o vetor em elementos.
size_type capacity() const	Número de elementos para os quais o vetor alocou memória. O valor de retorno de *capacity* é sempre maior ou igual a *size*.
bool empty() const	*Verdadeiro* se o tamanho do vetor for 0.

Tabela 1225 Os membros da classe *vector*. (*Continuação*)

Membro	Descrição
reference operator[](size_type n)	Retorna o *enésimo* elemento no repositório.
const_reference operator[](size_type n) const	Retorna uma representação em valor constante do enésimo elemento no repositório.
vector()	Cria um vetor vazio.
vector(size_type n)	Cria um vetor com *n* elementos.
vector(size_type n, const T & t)	Cria um vetor com *n* cópias do objeto *t*.
vector(const vetor&)	A construtora da cópia.
vector(InputIterator, InputIterator)	Cria um vetor com uma cópia de um intervalo.
~vector()	A função destrutora.
vector& operator=(const vector&)	O operador de atribuição.
void reserve(size_t n)	Se *n* for menor ou igual a *capacity*, esta chamada não terá efeito. Caso contrário, será uma solicitação de alocação de memória adicional. Se a solicitação for bem-sucedida, então a função definirá *capacity* como maior ou igual a *n*; caso contrário, *capacity* será inalterada. Em ambos os casos, a propriedade *size* do vetor permanecerá inalterada.
reference front()	Retorna o primeiro elemento no vetor.
const_reference front() const	Retorna uma representação de valor constante do primeiro elemento no vetor.
reference back()	Retorna o último elemento no vetor
const_reference back() const	Retorna uma representação de valor constante do último elemento no vetor.
void push_back(const T&)	Insere um novo elemento no final do vetor.
void pop_back()	Remove o último elemento do vetor.
void swap(vector&)	Permuta o conteúdo de dois vetores, desde que os vetores sejam de tipos compatíveis. Em outras palavras, se um vetor Z e Y têm ambos valores inteiros, você pode permutar o conteúdo dos vetores com o comando *swap*.
iterator insert(iterator pos, const T & x)	Insere *x* antes de *pos* em um vetor.
void insert(iterator pos, InputIterator f, InputIterator l)	Insere o intervalo (primeiro, último) antes de *pos* no vetor.
void insert(iterator pos, size_type n cons T & x)	Insere n cópias de *x* antes de *pos* dentro do vetor
iterator erase(iterator pos)	Apaga o elemento na posição *pos* do vetor.
iterator erase(iterator primeiro, iterator ultimo)	Apaga o intervalo [primeiro, ultimo) do vetor.
bool operator==(const vector&, const vector&)*	Testa se dois vetores são iguais. Essa é uma função global, não uma função-membro.
bool operator<(const vector&, const vector&)	Comparação lexicográfica. Essa é uma função global, não uma função-membro.

USANDO OUTRO PROGRAMA SIMPLES DE VETOR 1226

Na dica anterior você examinou as funções-membro que a classe *vector* suporta. Na Dica 1207, você criou um programa simples, *vetor1.cpp*, que usava várias funções-membro *vector* para manipular as informações dentro de um *vetor*. Antes de você aprender e usar a classe *bit_vector*, é importante que escreva outro programa que use as funções-membro da classe *vector* para manipular um *vetor*. O programa a seguir, *vetor2.cpp*, usa as funções-membro *reserve*, *max_size*, *resize* e *capacity* com um *vetor* contendo somente um único inteiro:

```
#include <iostream.h>
#include <vector.h>

using namespace std;
typedef vector<int> INTVECTOR;

void main(void)
 {
   // O vetor alocado dinamicamente começa com 0 elementos.
   INTVECTOR Vetor;

   // Adiciona um elemento ao final do vetor, um int com o valor 42.
   Vetor.push_back(42);

   // Exibe estatísticas sobre o vetor.
   cout << "O tamanho de Vetor é: " << Vetor.size() << endl;
   cout << "O tamanho máximo de Vetor é: " << Vetor.max_size()<< endl;
   cout << "a capacidade de Vetor é: " << Vetor.capacity() << endl;

   // Garante que há espaço para pelo menos 1000 elementos
   Vetor.reserve(1000);
   cout << endl << "Após reservar espaço para 1000 elementos:" << endl;
   cout << "O tamanho de Vetor é: " << Vetor.size() << endl;
   cout << "O tamanho máximo de Vetor é: " << Vetor.max_size()<< endl;
   cout << "A capacidade de Vetor é: " << Vetor.capacity() << endl;

   // Garante que há espaço para pelo menos 2000 elementos.
   Vetor.resize(2000);
   cout << endl << "Após redefinir o espaço para 2000 elementos:" << endl;
   cout << "O tamanho de Vetor é: " << Vetor.size() << endl;
   cout << "O tamanho máximo de Vetor é: " << Vetor.max_size()<< endl;
   cout << "A capacidade de Vetor é: " << Vetor.capacity() << endl;
 }
```

O programa *vetor2.cpp* primeiro inicializa um *vetor*. Segundo, o programa acrescenta um único elemento inteiro ao *vetor* e mostra informações sobre o tamanho atual, tamanho máximo e capacidade do *vetor*. Terceiro, o programa reserva armazenamento para 1.000 elementos e exibe as mesmas informações sobre o *vetor*. Finalmente, o programa redefine o tamanho do *vetor* para 2.000 elementos, depois exibe as informações sobre o *vetor*. O programa usa a função-membro *max_size*, que retorna o número máximo de elemento que o *vetor* pode conter. O programa também usa a função-membro *capacity* para retornar o número de elementos para os quais o *vetor* realmente alocou memória. Quando você compilar e executar o programa *vetor2.cpp*, sua tela exibirá a seguinte saída:

```
O tamanho do vetor é: 1
O tamanho máximo do vetor é: 1073741823
A capacidade do vetor é: 1
```

```
Após reservar armazenagem para 1000 elementos:
O tamanho do vetor é: 1
O tamanho máximo do vetor é: 1073741823
A capacidade do vetor é: 1000

Após redefinir a armazenagem para 2000 elementos:
O tamanho do vetor é: 2000
O tamanho máximo do vetor é: 1073741823
A capacidade do vetor é: 2000
C:\>
```

1227 COMPARANDO OS VETORES COM AS MATRIZES DE C

Como você viu, um *vetor* é similar a uma matriz em C. No entanto, existem diferenças significativas entre as matrizes e os *vetores*. Como regra, os *vetores* são ferramentas mais poderosas pelas seguintes razões:

- Seus programas podem dinamicamente realocar o tamanho de um *vetor* em qualquer ponto durante a execução do programa, como mostrado na dica anterior;
- Como os *vetores* usam iteradores, percorrer o *vetor* e acessar os elementos que o *vetor* contém é um processo mais poderoso que percorrer a matriz e acessar seus elementos. Por exemplo, as funções-membro *front*, *back*, *begin* e *end* oferecem modos úteis de acessar itens específicos dentro do *vetor* sem conhecer a localização exata deles.
- Embora os *vetores* aloquem capacidade, as matrizes de C alocam memória real. Uma matriz de 1.000 inteiros aloca 2.000 bytes de memória em sua declaração; um *vetor* que reserva 2.000 bytes de memória, mas armazena somente um único inteiro, usa somente 2 bytes de memória, como você viu na dica anterior (excluindo o gasto do objeto — o próprio objeto consome memória).
- Se você inadvertidamente colocar itens demais dentro de um *vetor*, a classe automaticamente realocará o tamanho do *vetor* para acomodar os elementos adicionais.
- Seus programas podem usar algoritmos para manipular os valores dentro dos *vetores* e das matrizes. No entanto, eles podem mais facilmente transferir valores entre os *vetores* e os outros tipos da Biblioteca de Gabaritos Padrões do que entre as matrizes e os tipos da Biblioteca de Gabaritos Padrões.

Em resumo, os *vetores* e as matrizes são muito similares. À medida que seus programas vão se tornando mais e mais complexos, é provável que você passe a usar mais *vetores* do que matrizes. No entanto, se você achar que está mais à vontade com as matrizes do que com os *vetores* e não precisa da flexibilidade adicional deles, não pense que precisa usar *vetores* em lugar das matrizes.

1228 COMPREENDENDO O REPOSITÓRIO DE SEQÜÊNCIA BIT_VECTOR

Nas Dicas 1225 até 1227, você trabalhou mais de perto com o tipo *vetor*. A versão atual da Biblioteca de Gabaritos Padrões também suporta o tipo *bit_vector*. Um *bit_vector* é basicamente um *vetor* do tipo *bool*. O Repositório de Seqüência *bit_vector* tem a mesma interface que um *vetor*. A principal diferença entre um *vetor* e um *bit_vector* é que os projetistas da Biblioteca de Gabaritos Padrões otimizaram a classe *bit_vector* para alcançar eficiência de espaço. Um *vetor* sempre requer pelo menos um byte por elemento, mas um *bit_vector* somente requer um bit por elemento.

É importante reconhecer que você precisa usar um iterador do tipo *bit_vector::iterator* com um *bit_vector*. A razão para a modificação nos tipos de iterador é simples — o *bit_vector* usa somente um bit para cada item, enquanto que um *vetor* usa somente um byte no mínimo. Se você tentasse usar um iterador *vetor* com um *bit_vector*, retornaria os próximos oito itens no *bit_vector* toda vez que incrementasse o iterator. Você implementará o tipo *bit_vector* dentro de seus programas, como segue:

```
bit_vector V(5);
V[0] = true;
V[1] = false;
V[2] = false;
```

```
V[3] = true;
V[4] = false;

for (bit_vector::iterator i = V.begin(); i < V.end(); ++i)
    cout << (*i ? '1' : '0');
cout << endl;
```

Nota: O Comitê de Padronização de C++ poderá remover o tipo bit_vector em uma futura versão da Biblioteca de Gabaritos Padrões. A razão de bit_vector ser uma classe separada, em vez de uma especialização de gabarito de vector<bool>, é porque criar uma especialização de gabarito com somente o tipo bool requereria especialização parcial de gabaritos. Após os compiladores que suportam especialização parcial tornarem-se mais comuns, uma especialização de vector<bool> poderá substituir bit_vector.

USANDO UM EXEMPLO BVECTOR SIMPLES 1229

Como você aprendeu na dica anterior, um *bit_vector* é um *vetor* que você usará especificamente para armazenar somente um único bit para cada elemento dentro do *vetor*. Como o *bit_vetor* usa somente um único bit, ele é capaz de armazenar somente valores *true* e *false*. Você também usará somente os iteradores *bit_vector::iterator* com *bit_vectors*. O programa a seguir, *bvetor1.cpp* altera o programa *vetor1.cpp* que você escreveu na Dica 1207 para usar um *bit_vector* em vez de um inteiro *vector*:

```
#include <iostream.h>
#include <bvector.h>

using namespace std;
const TAM_MATRIZ = 4;

void main(void)
 {
   // Vetor alocado dinamicamente inicia com 0 elementos.
   bit_vector Vetor(TAM_MATRIZ);

   // Inicializa a matriz para conter os membros
   // [100, 200, 300, 400]
   for (int cCadaItem = 0; cCadaItem < TAM_MATRIZ; cCadaItem++)
      if(cCadaItem>1)
          Vetor.push_back(cCadaItem - 2);
      else
         Vetor.push_back(cCadaItem);

   cout << "Primeiro elemento: " << Vetor.front()
        << endl;
   cout << "Último elemento: " << Vetor.back() << endl;
   cout << "Elementos no vetor: " << Vetor.size()
        << endl;

   // Excluindo o último elemento do vetor. Lembre-se que o
   // vetor tem sua base em 0, de modo que Vetor.end() na
   // verdade aponta 1 elemento além do final.
   cout << "Excluindo o último elemento." << endl;
   Vetor.erase(Vetor.end() - 1);
   cout << "O novo último elemento é: " << Vetor.back()  << endl;

   // Exclui o primeiro elemento do vetor.
   cout << "Excluindo o primeiro elemento." << endl;
   Vetor.erase(Vetor.begin());
   cout << "O novo primeiro elemento é: " << Vetor.front() << endl;
   cout << "Elementos no vetor: " << Vetor.size() << endl;
}
```

Quando você compilar e executar o programa *bvetor1.cpp*, sua tela exibirá a seguinte saída:

```
Primeiro elemento: 0
Último elemento: 1
Elementos no vetor: 4

Excluindo o último elemento.
O novo último elemento é: 0
Excluindo o primeiro elemento.
O novo primeiro elemento é: 1
Elementos no vetor: 2
C:\>
```

Nota: Nem o Visual C++ nem o Borland C++ 5.02 incluem o arquivo de cabeçalho bvector.h dentro da implementação padrão da Biblioteca de Gabaritos Padrões. Se você quiser usar o tipo bit_vector dentro de seus programas, precisará extrair o arquivo de cabeçalho do arquivo de inclusão da Biblioteca de Gabaritos Padrões e colocá-lo dentro do diretório include apropriado do seu compilador.

1230 COMPREENDENDO O TIPO LIST

Como você aprendeu, a Biblioteca de Gabaritos Padrões suporta uma variedade de repositórios. Um dos repositórios que você usará com mais freqüência e que deverá lhe parecer mais familiar, é o repositório *list* — uma lista ligada dupla (similar ao objeto *linked_list* que você criou em dicas anteriores). Isto é, *list* é um Repositório de Seqüência que suporta a navegação para frente e para trás, e a inserção e remoção de elementos em tempo linear, no início, no final ou no meio. A modificação de lista tem as três importantes propriedades a seguir:

- A inserção não invalida os iteradores para os elementos da lista
- A inserção de um elemento no centro da lista não invalida os iteradores para os elementos da lista
- A remoção de elementos somente invalida os iteradores que apontam para os elementos removidos

Seus programas podem alterar a ordem dos iteradores (isto é, a *list<T>::iterator* poderia ter um predecessor ou um sucessor diferente após uma operação de lista que ela fez antes), mas a classe não invalidará os próprios iteradores ou os fará apontar para elementos diferentes a não ser que a invalidação ou mutação seja explícita.

Observe que as listas ligadas simples, que suportam somente a varredura para a frente, também são úteis às vezes. Se você não precisar percorrer de trás para a frente, o tipo *slist* poderá ser mais eficiente que *list*. Você aprenderá mais sobre o tipo *slist* em dicas posteriores.

1231 COMPREENDENDO OS COMPONENTES GENÉRICOS DO REPOSITÓRIO LIST

Como você aprendeu, a Biblioteca de Gabaritos Padrões suporta o tipo repositório *list*. Você trabalhará com o tipo *list* de forma muito parecida ao trabalho como o objeto *linked_list* que você criou em dicas anteriores. Embora a classe *list* encapsule inteiramente o equivalente da classe *link_object*, você somente poderá acessar os membros da classe *list*, não a classe a partir da qual ela deriva.

A classe *list* pode ser de qualquer tipo. Seus programas podem crir listas que suportam tipos simples e listas que suportam tipos complexos. Por exemplo, a definição a seguir cria uma lista de inteiros simples, acrescenta diversos valores na lista e exibe os valores da lista:

```
list<int> L;

L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(),   ostream_iterator<int>(cout, " "));   // Exibe 1 2 0
```

O código cria uma *lista L*, acrescenta três valores na lista e os exibe. Observe que o último comando usa o algoritmo *copy* da Biblioteca de Gabaritos Padrões para copiar cada elemento na lista para o canal de saída *cout*, e até insere um espaço após cada elemento que ele copia no canal. O tipo *list* suporta um único tipo genérico,

T. Você não pode criar listas que suportem múltiplos tipos dentro de um único tipo, a não ser que primeiro encapsule esses tipos dentro de uma classe ou de uma estrutura.

CONSTRUINDO UM OBJETO LIST 1232

Como você aprendeu na Dica 1230, a Biblioteca de Gabaritos Padrões suporta o tipo lista ligada dupla *list*. Como o tipo *list* é uma classe, você inicializará listas usando uma das diversas construtoras que a Biblioteca de Gabaritos Padrões define para a classe *list*. A Biblioteca de Gabaritos Padrões define quatro construtoras diferentes para o tipo *list*, como mostrado aqui:

```
explicit list(void);
explicit list(size_type n, const T&, valor);
list(const list& PrimList);
list(const_iterator prim, const_iterator ultimo);
```

A primeira construtora especifica uma lista inicial vazia. A segunda construtora especifica uma repetição dos *n* elementos do valor *valor* (ou você pode atribuir um padrão). A terceira construtora (a construtora da cópia) instrui o compilador a inicializar a nova lista com uma cópia de outra lista *PrimList*. A última construtora especifica a seqüência [primeiro, último), que são dois iteradores de outro objeto *list*. A construtora copia todos os elementos entre *ultimo* e *primeiro* do objeto *list* original no objeto lista recém-construído. Todas as construtoras armazenam o objeto alocador *al* ou, para a construtora de cópia, o valor de retorno de *PrimList.get_allocator*, no dado-membro *allocator*. Após armazenar o objeto *allocator*, todas as quatro construtoras inicializam a lista.

Para compreender melhor as diferentes construtoras que a classe *list* da Biblioteca de Gabaritos Padrões suporta, considere o seguinte programa, *list1.cpp*:

```
#include <list.h>
#include <iostream.h>
#include <string.h>

using namespace std;
typedef list<string> LISTSTR;

// Experimenta cada uma das quatro construtoras
void main(void)
 {
   LISTSTR::iterator i;
   LISTSTR testa;                  // construtora padrão

   testa.insert(testa.end(), "um");
   testa.insert(testa.end(), "dois");

   LISTSTR testa2(testa);          // construtora de outra lista
   LISTSTR testa3(3, "três");        // construção com três elementos que
                                   contém o valor "três"
   LISTSTR testa4(++testa3.begin(),testa3.end());       // cria a partir de
   // parte de testa3

   // Imprime todos êles
   for (i = testa.begin(); i != testa.end(); ++i)
      cout << *i << " ";
   cout << endl;

   for (i = testa2.begin(); i != testa2.end(); ++i)
     cout << *i << " ";
   cout << endl;
   for (i = testa3.begin(); i != testa3.end(); ++i)
```

```
    cout << *i << " ";
  cout << endl;

  for (i = testa4.begin(); i != testa4.end(); ++i)
    cout << *i << " ";
  cout << endl;
}
```

Quando você compilar e executar o programa *list1.cpp*, sua tela exibirá a seguinte saída:

```
um dois
um dois
três três três
três três
C:\>
```

1233 INSERINDO OBJETOS NA LISTA

Como você viu na dica anterior, é possível construir a lista de vários modos — um com nenhuma inicialização e dois com inicialização. Além disso, você pode atribuir valores aos elementos de uma lista de vários modos importantes. Como aprenderá na dica 1236, você pode usar as funções-membro *push_back* e *push_front* para somar valores no objeto *list*. No entanto, você também pode usar a função-membro *insert* para adicionar valores no objeto *list*. Você usará a função-membro *insert* dentro de seus programas, como mostra os seguintes protótipos:

```
iterator insert(iterator it, const T& x);
void insert(iterator it, size_type n, const T& x);
void insert(iterator it, const_iterator prim, const_iterator ultimo);
```

Como você pode ver, o objeto *list* fornece três versões sobrecarregadas das funções-membro *insert*. Cada função-membro insere uma seqüência que os operandos restantes especificam antes do elemento para o qual o iterador *it* aponta na seqüência controlada. A primeira função-membro insere um único elemento com valor *x* e retorna um iterador que aponta para o elemento recém-inserido. A segunda função-membro insere uma repetição de *n* elementos do valor *x* (em outras palavras, cinco elementos do valor 2, por exemplo). A segunda função-membro insere uma repetição de *n* elementos do valor *x* (em outras palavras, cinco elementos do valor 2, por exemplo). A última função-membro insere a seqüência [primeiro, ultimo), começando na posição *it* do iterador.

O CD-ROM que acompanha este livro inclui o programa *insere_3.cpp*, que usa os três métodos *insert* para inserir elementos na lista.

1234 USANDO A FUNÇÃO-MEMBRO ASSIGN

Na dica anterior mostrou, você aprendeu como seus programas podem usar a função-membro *insert* para inserir elementos em uma lista. Seus programas também podem usar a função-membro *assign* para atribuir valores aos elementos existentes dentro de uma lista. Em vez de inserir novos elementos e excluir os antigos elementos, você pode usar *assign* para substituir uma série de elementos dentro de uma lista com um único comando. Você usará a função-membro *assign* dentro de seus programas em uma de suas duas formas sobrecarregadas, como a seguir nesses protótipos:

```
void assign(const_iterator prim, const_iterator ultimo);
void assign(size_type n, const T& x = T());
```

A primeira função-membro substitui a seqüência para a qual **this* aponta pela seqüência [prim, ultimo). A segunda função-membro substitui a sequência para a qual **this* aponta por uma repetição de *n* elementos do valor *x*. Por exemplo, o seguinte comando substitui os próximos quatro itens dentro de uma lista com o valor 1:

```
listOutra.assign(4, 1);
```

Dependendo de onde o iterador aponta atualmente dentro da lista, a atribuição poderia iniciar no início ou no meio da lista. Se você tentar atribuir um intervalo de valores a uma série de elementos da lista que excede o final da lista, a atribuição acrescentará elementos adicionais no final da lista.

USANDO AS FUNÇÕES-MEMBRO REMOVE E EMPTY 1235

A dica anterior mostrou como seus programas podem usar a função-membro *assign* para substituir uma série de elementos dentro de uma lista. Quando você trabalha com listas, precisa freqüentemente remover elementos além de reatribuir valores aos elementos dentro da lista. A classe *list* suporta duas funções-membro *remove* sobrecarregadas que você pode usar dentro de seus programas para remover elementos a partir de dentro de uma lista. Você usará as duas funções-membro *remove* sobrecarregadas, a seguir nesses protótipos:

```
iterator erase(iterator it);
iterator erase(iterator prim, iterator ultimo);
```

A primeira função-membro remove o elemento da sequência controlada para a qual o iterador *it* aponta. A segunda função-membro remove os elementos da sequência controlada no intervalo [prim, ultimo). Ambas as funções-membro *remove* retornarão um iterador que designa o primeiro elemento restante além de quaisquer elementos que as funções removeram, ou *end* se esse elemento não existir.

Ocasionalmente, você pode remover todos os elementos a partir de dentro de uma lista. Para proteger seus programas de tentar exibir ou manipular uma lista vazia, seus programas devem checar as remoções para determinar se quaisquer elementos permanecem dentro da lista. Para fazer isso, seus programas devem usar a função-membro *empty*, cujo protótipo é mostrado aqui:

```
bool empty(void) const;
```

A função-membro *empty* retornará *True* para uma seqüência controlada vazia. Para compreender melhor como seus programas podem usar as funções-membro *remove* e *empty*, bem como a função-membro *assign* que a dica anterior discutiu, considere o programa *list_are.cpp* no CD-ROM que acompanha este livro, o qual usa todas as três funções. O programa cria duas listas, listUm, e listOutro. O programa então atribui três valores para o repositório listUm e atribui um único valor ao repositório listOutro. Em seguida, o programa usa o método *assign* para copiar os três elementos em *listUm* em *listOutro* e sobrescreve o elemento que o comando anterior atribuiu. Após exibir o repositório *listOutro*, o programa atribui o valor um a todos os elementos da lista, o que substitui os valores de elemento anteriores. O programa então exibe o repositório *listOutro* novamente. Em seguida, o programa apaga o primeiro elemento na lista *listOutro* e exibe a lista de novo. Finalmente, o programa apaga todos os elementos na lista *listOutro* e gera a mensagem "Tudo terminado!".

PERCORRENDO O OBJETO LIST 1236

Na dica anterior, o programa *list_are.cpp* usou as funções-membro *begin* e *end* com um iterador *list* para percorrer a lista. Seus programas também podem usar as funções-membro *front* e *back* para controlar a posição de um iterador dentro de uma lista. A função-membro *front* retorna uma referência para o primeiro elemento da sequência controlada. A função-membro *back* retorna uma referência para o último elemento da seqüência controlada.

Quando seus programas usam as funções *front* e *back*, você também pode querer usar as funções *push* e *pop*, por questões de consistência. Lembre-se de que as funções *push* e *pop* colocam valores dentro da lista na frente e atrás. A função-membro *pop_back* remove o último elemento da sequência controlada. A função-membro *pop_front* remove o primeiro elemento da sequência controlada. Todas essas funções requerem que a sequência controlada não esteja vazia. A função-membro *push_front* insere um elemento com o valor *x* no início da seqüência controlada. A função-membro *push_back* insere um elemento com o valor *x* no final da seqüência controlada. Para compreender melhor o processamento que seus programas podem efetuar usando *front*, *back* e suas funções de atribuição associadas, considere o programa *frntback.cpp*, mostrado aqui:

```
#include <list.h>
#include <list>
#include <string.h>
#include <iostream.h>

using namespace std;
typedef list<string> LISTSTR;

void main(void)
```

```
{
  LISTSTR teste;

  teste.push_back("trás");
  teste.push_front("meio");
  teste.push_front("frente");
  cout << teste.front() << endl;      // frente
  cout << teste.back() << endl;       // trás
  teste.pop_front();
  teste.pop_back();
  cout << teste.front() << endl;      // meio
}
```

Quando você compilar e executar o programa *frntback.cpp*, sua tela exibirá a seguinte saída:

```
frente
trás
meio
C:\>
```

1237 COMPREENDENDO O TIPO SLIST

Como você aprendeu, a Biblioteca de Gabaritos Padrões suporta o tipo *list*, uma lista ligada dupla. A Biblioteca de Gabaritos Padrões também suporta o tipo *slist*, que é uma lista ligada simples que vincula cada elemento com o próximo na lista, mas não o elemento anterior. Isto é, um *slist* é um Repositório de Seqüência que permite percorrer para a frente mas não para trás, e permite a inserção e remoção de elementos em tempo constante. Exatamente como com a modificação *list*, a modificação *slist* tem as três seguintes propriedades importantes:

- A inserção não invalida os iteradores nos elementos da lista
- A inserção dentro da lista, em vez do final, não invalida os iteradores para os elementos da lista
- A remoção dos elementos da lista invalida somente os iteradores que apontam para os elementos removidos

Seus programas podem mudar o ordenação dos iteradores (isto é, *slist<T>::iterator* poderá ter um predecessor ou um sucessor diferente após uma operação de lista que tinha antes), mas a mudança não invalidará os próprios iteradores ou fará com que apontem para elementos diferentes, a não ser que a invalidação ou a mutação seja explícita.

A principal diferença entre *slist* e *list* é que os iteradores *list* são *bidirecionais*, enquanto que os iteradores *slist* são do tipo *forward* (apenas para a frente). A diferença nos tipos de iteradores significa que *slist* é menos versátil que *list*. No entanto, frequentemente, os iteradores *bidirecionais* não são necessários. Você deverá usar *slist* a não ser que precise da funcionalidade adicional de *list*, pois as listas ligadas simples são menores e mais rápidas do que as listas ligadas duplas.

Nota: Como com bit_vector, nem o Visual C++ nem o Borland C++ 5.02 inclui o arquivo de cabeçalho para o tipo slist.

1238 COMPREENDENDO AS INSERÇÕES DENTRO DE UM REPOSITÓRIO DE SEQÜÊNCIA SLIST

Na dica anterior você aprendeu sobre a seqüência *slist*. Como todo outro Repositório de Seqüência, *slist* define as funções-membro *insert* e *erase*. No entanto, usar essas funções-membro de forma descuidada pode resultar em programas excessivamente lentos. O problema é que o primeiro argumento de *insert* é um iterador *pos*, e *insert* coloca os novos elementos antes de *pos*. Em outras palavras, a função *insert* precisa encontrar o iterador imediatamente antes do iterador *pos*. Encontrar o iterador imediatamente antes de *pos* é uma operação de tempo constante para *list* porque *list* tem iteradores *bidirecionais*. No entanto, para *slist*, *insert* precisa percorrer a lista do início até *pos* para encontrar esse iterador. Em outras palavras, *insert* e *erase* são operações lentas em qualquer lugar, exceto perto do início de *slist*.

A classe *slist* fornece as funções-membro *insert_after* e *erase_after*, que são operações de tempo constante; você deve usar *insert_after* e *erase_after* sempre que possível. Se você achar que *insert_after* e *erase_after* não são adequadas para as suas necessidades e que você normalmente precisa usar *insert* e *erase* no meio da lista, deve provavelmente usar *list* em vez de *slist*.

COMPREENDENDO O REPOSITÓRIO DEQUE 1239

Um *deque* é muito parecido com um *vetor* porque é um Repositório de Seqüência que permite que seus programas acessem elementos aleatoriamente e efetua inserções em tempo constante, bem como remove elementos no final da seqüência. Um *deque* também permite que seus programas efetuem inserção de tempo linear e remoção de elementos no meio da seqüência.

O principal modo no qual um *deque* difere de um *vetor* é que um deque suporta inserção e remoção de elementos em tempo constante no início da seqüência (além do suporte de *vetor* para essas atividades no final da seqüência). Adicionalmente, um *deque* não tem quaisquer funções-membro similares às funções *capacity* e *reserve* de um *vetor*, e não fornece quaisquer uma das garantias da validade do iterador que a Biblioteca de Gabaritos Padrões associa com *capacity* e *remove*. Você declarará e usará os objetos *deque* dentro de seus programas, como mostrado aqui:

```
deque<int> Q;
Q.push_back(3);                    // coloca 3 no final
Q.push_front(1);                   // coloca 1 na frente
Q.insert(Q.begin() +1, 2);         // insere 2 no meio
Q[2] = 0;                          // define o terceiro elemento com 0
copy(Q.begin(), Q.end(), ostream_iterator<int>(cout, " "));
// Os valores que são impressos são 1 2 0
```

As funções-membro *push_back*, *push_front* e *insert* executam as mesmas tarefas com um *deque* que executam com um *vetor*. Adicionalmente, o algoritmo *copy* permite que seus programas copiem saída diretamente para o canal de saída, exatamente como você já fez antes com o objeto *list*.

USANDO O REPOSITÓRIO DEQUE 1240

A dica anterior mostrou que o repositório *deque* é similar ao repositório *vetor* e ao repositório *list*. Na verdade, *deque* usa elementos de ambos os tipos de repositórios. Como você pode esperar, o repositório *deque* inclui suporte para algumas funções-membro no estilo *vetor* e algumas no estilo *list*. Duas das funções-membro que o repositório *deque* suporta são a função-membro *swap* (do repositório no estilo *vetor*) e a função-membro *assign* (do repositório no estilo *list*). Você usará a função-membro *swap* e a função-membro *assign* dentro de seus programas, como mostra os seguintes protótipos:

```
void assign(const_iterator prim, const_iterator ultima);
void assign(size_type n, const T& x = T());
void swap(deque& dq);
```

A primeira função-membro *assign* substitui a seqüência para a qual **this* aponta com a seqüência [prim, ultima). A segunda função-membro *assign* substitui a seqüência para a qual **this* aponta com uma repetição de *n* elementos do valor *x*. A função-membro *swap* permuta os conteúdos entre **this* e *dq*. Para compreender melhor o processamento dessas funções-membro *deque*, considere o seguinte programa, *deque1.cpp*:

```
#include <deque.h>
#include <iostream.h>

using namespace std;
typedef deque<char> CHARDEQUE;

void imprime_conteudo(CHARDEQUE deque, char*);

void main(void)
 {
   CHARDEQUE  a(3, 'A');          // cria a com 3 As
```

```
    CHARDEQUE  b(4, 'B');          // cria b com 4 Bs.

    imprime_conteudo(a, "a"); // imprime o conteúdo
    imprime_conteudo(b, "b");
    a.swap(b);                // permuta a e b
    imprime_conteudo(a, "a");
    imprime_conteudo(b, "b");
    a.swap(b);                // permuta novamente
    imprime_conteudo(a, "a");
    imprime_conteudo(b, "b");
    a.assign(b.begin(),b.end()); // atribui b a a
    imprime_conteudo(a, "a");
    a.assign(b.begin(),b.begin()+2);       // atribui os dois primeiros
    // itens de b para a
    imprime_conteudo(a, "a");
    a.assign(3, 'Z');                  // atribui 3 'Z's a a
    imprime_conteudo(a, "a");
  }

// Função para imprimir o conteúdo do deque
void imprime_conteudo(CHARDEQUE deque, char *nome)
  {
    CHARDEQUE::iterator pdeque;

    cout << "O conteúdo de " << nome << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
      cout << *pdeque << " ";
    cout<< endl;
  }
```

Quando você compilar e executar o programa *deque1.cpp*, sua tela exibirá a seguinte saída:

```
O conteúdo de a : A A A
O conteúdo de b : B B B B
O conteúdo de a : B B B B
O conteúdo de b : A A A
O conteúdo de a : A A A
O conteúdo de b : B B B B
O conteúdo de a : B B B B
O conteúdo de a : B B
O conteúdo de a : Z Z Z
C:\>
```

1241 USANDO AS FUNÇÕES-MEMBRO ERASE E CLEAR

Na Dica 1239 você aprendeu sobre o repositório *deque* e sua similaridade com o repositório *vetor*. Na dica anterior, você aprendeu como usar as funções-membro *swap* e *assign* com o repositório *deque*. Como sabe, uma das atividades mais comuns que você executará com uma matriz, *vector* ou outra lista de elemento é remover elementos ou remover o valor dos elementos dentro da lista. O repositório *deque* fornece as funções-membro *erase* e *clear* para lhe ajudar a gerenciar elementos individuais e o repositório inteiro. As funções-membro *erase* apagam um único elemento ou um intervalo de elementos. A função-membro *clear* apaga todos os elementos dentro do *deque*. Você usará as funções-membro dentro de seus programas, como mostrado nos seguintes protótipos:

```
iterator erase(iterator iter);
iterator erase(iterator prim, iterator ultimo);
void clear(void) const;
```

A primeira função-membro *erase* remove o elemento do repositório para o qual *iter* aponta. A segunda função-membro *erase* remove os elementos do repositório no intervalo [prim, ultimo). Ambas retornam um iterator que designa o primeiro elemento restante além de quaisquer elementos que a função-membro removeu, ou *end()* se nenhum elemento existir. Remover *N* elementos causa *N* chamadas destrutoras e uma atribuição para cada ele-

mento entre o ponto de inserção e o final mais próximo da seqüência. Remover um elemento em uma das duas pontas invalida somente os iteradores e referências que designam os elementos apagados. Caso contrário, apagar um elemento invalida todos os iteradores e referências. A função-membro *clear* chama *erase(begin(), end())*. Para compreender melhor o processamento que essas funções-membro executam, considere o programa *erase_de cpp* no CD-ROM que acompanha este livro, que usa todas as três.

USANDO O OPERADOR DE MATRIZ [] COM UM DEQUE 1242

Como você aprendeu, um *deque* é similar a um *vetor* e, portanto, similar a uma matriz. Como o *deque* é similar a uma matriz, você poderia esperar que seus programas pudessem usar o operador de matriz [] para acessar elementos específicos dentro de um *deque*. No entanto, os programas que você escreveu até aqui usaram iteradores com objetos *deque*, em vez do operador matriz em índice. Como se verá, você pode usar um iterador, o operador matriz, ou ambos, para acessar um *deque* dentro de um determinado programa. Seus programas também podem usar a função-membro *at* para mudar de índices de matrizes para iteradores.

A função-membro *operator[]* retorna uma referência ao elemento da seqüência na posição *pos*. Se essa posição for inválida, o comportamento da função será imprevisível. A função-membro *at* retorna uma referência ao elemento da seqüência controlada na posição *pos*. Se essa posição for inválida, a função lançará um objeto da classe exceção *out_of_range*. A função-membro *empty* retorna True para uma seqüência controlada vazia. O programa a seguir, *deq_arr.cpp*, usa um iterador, o operador matriz e a função-membro *at* para acessar um *deque*:

```
#include <deque.h>
#include <iostream.h>

using namespace std;
typedef deque<char> CHARDEQUE;
void imprime_conteudo(CHARDEQUE deque, char*);

void main(void)
 {
   CHARDEQUE  a;     // cria um deque vazio a

   if(a.empty())     // checa se ele está vazio
     cout << "a está vazio" << endl;
   else
     cout << "a não está vazio" << endl;

   a.push_back('A');     //inset A, B, C e D em a
   a.push_back('B');
   a.push_back('C');
   a.push_back('D');
   if(a.empty())      // checa novamente se a está vazio
       cout << "a está vazio" << endl;
   else
       cout << "a não está vazio" << endl;
   imprime_conteudo(a,"a");     // imprime o conteúdo

   cout << "O primeiro elemento de a é " << a[0] << endl;
   cout << "O primeiro elemento de a é " << a.at(0) << endl;
   cout << "O último elemento de a é " << a[a.size()-1] << endl;
   cout << "O último elemento de a é " << a.at(a.size()-1)  << endl;
 }

// Função para imprimir o conteúdo do deque
void imprime_conteudo(CHARDEQUE deque, char *nome)
 {
   CHARDEQUE::iterator pdeque;
```

```
    cout << "O conteúdo de "<< nome << " : ";
    for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
      cout << *pdeque <<" ";
    cout << endl;
  }
```

Quando você compilar e executar o programa *deq_arr cpp*, sua tela exibirá a seguinte saída:

```
a está vazia
a não está vazia
Conteúdo de a: A B C D
O primeiro elemento de a é A
O primeiro elemento de a é A
O último elemento de a é D
O último elemento de a é D
C:\>
```

1243 USANDO OS ITERADORES REVERSE COM UM DEQUE

Em dicas anteriores, você usou ambos os iteradores e o operador *matriz* para manipular e retornar os valores dentro de um *deque*. No entanto, os *deques* também suportam os iteradores *reverse*. Como você aprendeu, um iterador *reverse* é um iterador que aponta do final do *deque* para a frente. Em outras palavras, quando você incrementar um iterador *reverse*, ele se moverá para baixo no *deque*, não para cima como faz um iterador do tipo *forward*. A Figura 1243 mostra como o movimento de um iterador *reverse* é diferente do movimento de um iterador *forward*.

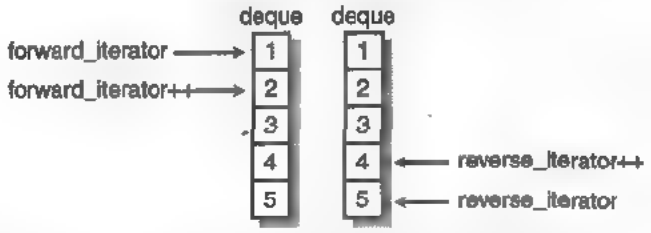

Figura 1243 O iterador *reverse* move-se para baixo em vez de para cima no *deque*.

Dentro de seus programas, você pode usar as funções-membro *rbegin* e *rend* para obter um iterador *reverse* para um *deque*. A função-membro *rbegin* retorna um iterador *reverse* que aponta para logo após o final da seqüência controlada. Portanto, *rbegin* designa o início da seqüência inversa. A função-membro *rend* retorna um iterador *reverse* que aponta para o primeiro elemento da seqüência, ou imediatamente após o final de uma seqüência vazia. Portanto, ela designa o final da seqüência inversa. O programa a seguir, *rev_iter cpp*, usa *rbegin* e *rend* para navegar um *deque* em ordem inversa:

```
#include <iostream.h>
#include <deque.h>.

#include <iostream.h>

using namespace std;
typedef deque<int> INTDEQUE;

void main(void)
 {
   // Cria A e preenche-o com elementos 1,2,3,4 e 5 usando a
   // função push_back

   INTDEQUE A;
   A.push_back(1);
   A.push_back(2);
   A.push_back(3);
```

```
    A.push_back(4);
    A.push_back(5);

    // Agora imprima o conteúdo em ordem reversa usando
    // reverse_iterator e as funções rbegin() e rend()

    INTDEQUE::reverse_iterator rpi;
    for(rpi = A.rbegin(); rpi != A.rend(); rpi++)
       cout << *rpi << " ";
    cout<< endl;
}
```

O programa cria o *deque* em branco *A*, depois atribui cinco valores a ele. Em seguida, o programa define um iterador *reverse*. Finalmente, o programa usa o iterador *reverse* para exibir o conteúdo do *deque* em ordem inversa. Quando você compilar e executar o programa *rev_iter.cpp*, sua tela exibirá a seguinte saída:

```
5 4 3 2 1
C:\>
```

GERENCIANDO O TAMANHO DO DEQUE 1244

Em dicas recentes, você usou funções-membro para criar e gerenciar os *deques*. Como aprendeu, a classe *deque* tem características similares a ambas as classes *vetor* e *list*. Um dos recursos mais poderosos dos *vetores*, que você também pode usar com objetos *deque*, é o gerenciamento do tamanho do objeto. O tipo *deque* fornece três funções-membro que você pode usar para gerenciar o tamanho de um *deque*, como mostrado aqui:

```
size_type size(void) const;
void resize(size_type NewSize, T x = T());
size_type max_size(void) const;
```

A função-membro *size* retorna o tamanho (isto é, o número atual de elementos) da seqüência. A função-membro *resize* muda o tamanho para o número de elementos que *NewSize* especifica. Se *resize* precisar tornar a seqüência controlada maior, ela acrescentará elementos com o valor *x*. Se *resize* não fornecer nenhum valor, o valor padrão dependerá no tipo do objeto *deque*. Por exemplo, se o *deque* for um *deque* de *chars*, o padrão será um branco. Se o *deque* for um *deque* de inteiros, o padrão será zero. A função-membro *max_size* retorna o tamanho da maior seqüência que o objeto pode controlar. Para compreender melhor o processamento que as funções-membro *size*, *resize* e *max_size* executam com os objetos *deque*, considere o programa *deq_size.cpp*, contido no CD-ROM que acompanha este livro.

Seu programa cria um objeto *deque* para conter elementos *char*. O programa então atribui quatro caracteres ao objeto *deque*. Após atribuir os quatro caracteres ao *deque*, o programa imprime o *deque*, seu tamanho máximo, e seu tamanho atual. O programa então redefine o *deque* para dez caracteres e preenche do quinto até o décimo elemento com *X*. O programa exibe o *deque* novamente, bem como seu tamanho atual. O programa então redefine o tamanho do *deque* para cinco elementos e imprime esses cinco elementos. Finalmente, o programa imprime o tamanho atual do objeto *deque* e o tamanho máximo para mostrar que o tamanho máximo permanece inalterado.

COMPREENDENDO O OBJETO MAP 1245

Um *map* é um Repositório Associativo Classificado que associa objetos do tipo *Key* com objetos do tipo *Data*. Ao contrário dos Repositórios de Seqüência com os quais você trabalhou anteriormente, um *map* é um Repositório Associativo em Par, o que significa que o tipo de seu valor é *pair<const Key, Data>*. Além de ser um Repositório Associativo em Par, um *map* é um Repositório Associativo Único, o que significa que dois elementos não têm a mesma chave.

Um *map* lhe permite inserir um novo elemento em um *map* sem invalidar os iteradores que apontam para os elementos existentes. Se você apagar um elemento de um conjunto também não invalidará quaisquer iteradores, exceto, é claro, aqueles iteradores que realmente apontam para o elemento que você está apagando.

Os *maps* diferem dos repositórios com os quais você já trabalhou antes de dois modos: ele classifica os elementos automaticamente e usa uma chave além do próprio elemento. Por exemplo, os *maps* são úteis para man-

ter uma representação numérica classificada de uma série de strings, tal como uma lista de funcionários organizada por número de PIS. Você trabalhará com *maps* nas próximas dicas.

1246 UM EXEMPLO SIMPLES DE MAP

Como você aprendeu na dica anterior, seus programas podem usar *maps* para manter conjuntos de objetos em pares. Para compreender melhor como seus programas podem usar *maps* para gerenciar as informações, considere o programa a seguir, *map1.cpp:*

```
#include <map.h>
#include <iostream.h>
#include <string.h>

using namespace std;
class ltstr
 {
  public:
    bool operator()(const char* s1, const char* s2) const
      { return (strcmp(s1, s2) < 0);}
 };

void main(void)
 {
  map<const char*, int, ltstr> meses;

  meses["Janeiro"] = 31;
  meses["Fevereiro"] = 28;
  meses["Março"] = 31;
  meses["Abril"] = 30;
  meses["Maio"]  = 31;
  meses["Junho"] = 30;
  meses["Julho"] = 31;
  meses["Agosto"]   = 31;
  meses["Setembro"] = 30;
  meses["Outubro"]  = 31;
  meses["Novembro"] = 30;
  meses["Dezembro"] = 31;

  cout << "Junho -> " << meses["Junho"] << endl;
  map<const char*, int, ltstr>::iterator cur = meses.find("Junho");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "O anterior (em ordem alfabética) é " << (*prev).first << endl;
  cout << "O próximo (em ordem alfabética) é "  << (*next).first << endl;
 }
```

O programa *map1.cpp* cria um *map* simples de meses. Ele depois exibe o valor numérico de Junho dentro do *map* (6) classificado. Em seguida, o programa pesquisa no *map* o mês de Junho e exibe os meses cujos nomes precedem imediatamente e seguem Junho (isto é, em ordem alfabética: Julho e Março) na tela. Quando você compilar e executar o programa *map1.cpp*, sua tela exibirá a seguinte saída:

```
Junho -> 6
O anterior (na ordem alfabética) é Julho
O próximo (na ordem alfabética) é Março
C:\>
```

USANDO FUNÇÕES MEMBRO PARA GERENCIAR O MAP 1247

Vimos na Dica 1245 que seus programas podem usar os repositórios *map* para manter listas classificadas de objetos. Na dica anterior, você usou a função-membro *find* para localizar um valor dentro do *map*. Você então atribuiu dois iteradores ao *map* e usou os iteradores para exibir os valores mapeados, em vez dos valores-chave, dentro do *map*. Além da função-membro *find*, seus programas podem usar as funções-membro *end* e *insert* para manter informações dentro de seus *maps*. Você usará todas as três funções, como mostrado nos seguintes protótipos:

```
// Key é o tipo de dado do argumento gabarito #1 para map
iterator map::find(const Key& key);

iterator map::end(void);
pair<iterator, bool> map::insert(const valor_tipo& x);
```

A função *end* retorna um iterador que aponta para um além do final de uma sequência. *Find* retorna um iterador que designa o primeiro elemento cuja chave de classificação é igual a *key*. Se esse elemento não existir, o iterador será igual a *end()*. Se a chave não existir ainda, *insert* a acrescentará à sequência e retornará *pair<iterator, true>*. Se a chave já existir, *insert* não acrescentará a chave à sequência e, em vez disso retornará *pair<iterator, false>*. O programa a seguir, *map_ints.cpp*, cria um *map* de inteiros para strings. Neste caso, o mapeamento é de dígitos para seus equivalentes em string (1 torna-se "Um", 2 torna-se "Dois", e assim por diante). O programa lê um número do usuário, encontra a palavra equivalente para cada dígito (usando o mapa) e imprime o número como uma série de palavras. Por exemplo, se o usuário digitar 25463, o programa responderá com: Dois Cinco Quatro Seis Três. O programa *map_ints.cpp*, como mostrado aqui, usa as funções-membro *find*, *end* e *insert*:

```
#include <map.h>
#include <iostream.h>
#include <string.h>

using namespace std;
typedef map<int, string, less<int>> INT2STRING;

void main(void)
 {
   // Cria um map de ints para strings
   INT2STRING Mapa;
   INT2STRING::iterator Iterador;
   string aString = "";
   int indice, continue = 1;

   // Preenche Mapa com os dígitos 0 - 9, cada um mapeado
   // para seu correspondente string
   // Nota: value_type é um par para maps...
   Mapa.insert(INT2STRING::value_type(0,"Zero"));
   Mapa.insert(INT2STRING::value_type(1,"Um"));
   Mapa.insert(INT2STRING::value_type(2,"Dois"));
   Mapa.insert(INT2STRING::value_type(3,"Três"));
   Mapa.insert(INT2STRING::value_type(4,"Quatro"));
   Mapa.insert(INT2STRING::value_type(5,"Cinco"));
   Mapa.insert(INT2STRING::value_type(6,"Seis"));
   Mapa.insert(INT2STRING::value_type(7,"Sete"));
   Mapa.insert(INT2STRING::value_type(8,"Oito"));
   Mapa.insert(INT2STRING::value_type(9,"Nove"));
   // Lê um número do usuário e imprime-o por extenso
   while (continue)
    {
      cout <<"Digite \"s\" para sair, ou digite um número: ";
       cin >> aString;
       if(aString == "s")
```

```
        continue = 0;
      // extrai cada dígito da string, encontra sua
      // entrada correspondente no map (a palavra
      // equivalente) e imprime-o
      for( indice = 0; indice < aString.length(); indice++)
       {
         Iterador = Mapa.find(aString[indice] - '0');
         if(Iterador != Mapa.end())       // é 0 - 9
           cout << (*Iterador).second << " ";
         else                             // não 0 - 9
           cout << "[err] ";
       }
      cout << endl;
   }
}
```

O programa cria um *map* simples de números para objetos string. Quando encontrar um número dentro do *map*, o programa exibirá o equivalente string do número (por exemplo, 21 exibe dois um) O programa cria o *map*, depois entra em um laço infinito até que o usuário selecione a tecla 's', quando então o programa termina. Quando você compilar e executar o programa *map_ints.cpp*, sua tela exibirá a seguinte saída (desde que você insira os mesmos valores nos prompts):

```
Digite "s" para sair, ou digite um número: 911
Nove Um Um
Digite "s" para sair, ou digite um número: 1500
Um Cinco Zero Zero
Digite "s" para sair, ou digite um número: 4995
Quatro Nove Nove Cinco
Digite "s" para sair, ou digite um número: s
C:\>
```

1248 CONTROLANDO O TAMANHO E O CONTEÚDO DO MAP

Na dica anterior, você usou a função *insert* para acrescentar dez pares de valores a um *map*. À medida que seus programas forem se tornando mais complexos, algumas vezes você precisará apagar ou substituir os elementos dentro de um *map*, bem como algumas vezes precisará determinar o maior tamanho possível do objeto *map*. A implementação da Biblioteca de Gabaritos Padrões do tipo *map* lhe permite executar todas essas ações no seu *map*. Na verdade, as quatro funções-membro que você usa para efetuar essas atividades são similares àquelas que você usou anteriormente com outros objetos da Biblioteca de Gabaritos Padrões, como segue:

```
size_type max_size(void) const;
void clear(void) const;
bool empty(void) const;
iterator erase(iterator prim, iterator ultimo);
```

A função *max_size* lhe permite definir o limite superior no tamanho de um *map*. A função *clear* lhe permite apagar todos os elementos dentro de um *map*. Similarmente, a função *empty* permite que seus programas determinem se um *map* está ou não vazio de todos os seus elementos. Finalmente, a função *erase* lhe permite limpar um intervalo de elementos dentro de um *map*. (Observe que usar o comando *clear* é idêntico a chamar *erase* com [*erase(map.begin, map.end)*] — usar *clear* simplesmente torna mais óbvio o que o processamento do comando é). Para compreender melhor o processamento que essas funções-membro executam, considere o programa *map_mon2.cpp*, contido no CD-ROM que acompanha este livro, o que novamente cria um map dos meses. No entanto, o programa *map_mon2.cpp* limpa o *map* e o repreenche com um *map* dos nomes dos dias da semana. O programa *map_mon2.cpp* cria um *map* de strings para inteiros e o preenche primeiro com um *map* dos nomes dos meses com os números dos meses. O programa então esvazia e repreenche o *map* com um *map* de nomes dos dias da semana com inteiros correspondentes.

COMPREENDENDO OS SETS 1249

Um *set* (conjunto) é um Repositório Associativo Classificado que armazena objetos do tipo *key*. Além de sua natureza classificada, um *conjunto* é um Repositório Associativo Simples, o que significa que seu tipo de valor, bem como seu tipo-chave é *key*. Finalmente, um *conjunto* também é um Repositório Associativo, o que significa que não existem dois elementos iguais no repositório. Um *set* basicamente é um *map* com somente um único tipo, em vez de um tipo *value* e um tipo *key*. Como você aprendeu, *keys* são constantes e não alteráveis. Quando você trabalha com *sets*, cada elemento é inalterável após você inicialmente atribuir seu valor. Seu programa precisa, em vez disso, excluir o valor e acrescentar um elemento de substituição ao *set*.

Os tipos *set* e *multiset* são particularmente bem adequados para os algoritmos *set* que a Biblioteca de Gabaritos Padrões inclui: *set_union*, *set_intersection*, *set_difference* e *set_symmetric_difference*. Existem duas razões para isso. Primeiro, os algoritmos *set* requerem que seus argumentos sejam intervalos classificados e, como *set* e *multiset* são Repositórios Associativos Classificados, cada repositório sempre armazena seus elementos em ordem ascendente. Segundo, o intervalo de saída desses algoritmos é sempre classificado, e inserir um intervalo classificado em um *set* ou *multiset* é uma operação rápida: os requisitos do Repositório Associativo Classificado Único e do Repositório Associativo Múltiplo Classificado garantem que inserir um intervalo toma somente tempo linear se o intervalo já estiver classificado.

Um *set* lhe permite inserir um novo elemento em um *set* sem invalidar os iteradores que apontam para os elementos existentes. Um *set* também lhe permite apagar um elemento a partir de um *set* sem invalidar quaisquer iteradores, exceto, é claro, aqueles iteradores que realmente apontam para o elemento que você estiver apagando. O programa a seguir, *set1.cpp*, cria dois *sets* simples, compara seus valores e cria um terceiro *set*, como mostrado aqui:

```
#include <set.h>
#include <iostream.h>

using namespace std;
class ltstr
 {
  public:
    bool operator()(const char* s1, const char* s2) const
      { return (strcmp(s1, s2) < 0);}
 };

void main(void)
 {
   const int N = 6;
   const char* a[N] = {"aaa", "bbb", "ccc", "ddd", "eee", "fff"};
   const char* b[N] = {"ggg", "hhh", "eee", "iii", "ccc", "aaa"};

   set<const char*, ltstr> A(a, a + N);
   set<const char*, ltstr> B(b, b + N);
   set<const char*, ltstr> C;
   cout << "Set A: ";
   copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
   cout << endl;
   cout << "Set B: ";
   copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
   cout << endl;
   cout << "União: ";
   set_union(A.begin(), A.end(), B.begin(), B.end(), ostream_
      iterator<const char*>(cout, " "), ltstr());
   cout << endl;
   cout << "Intersecção: ";
   set_intersection(A.begin(), A.end(), B.begin(),
            B.end(), ostream_iterator<const char*> (cout, " "), ltstr());
   cout << endl;
   set_difference(A.begin(), A.end(), B.begin(),
            B.end(),inserter(C, C.begin()), ltstr());
   cout << "Set C (diferença de A e B): ";
```

```
    copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
    cout << endl;
  }
```

O programa *set1.cpp* cria um *set* de união, um *set* de intersecção e um *set* de diferença. O *set* de união contém todos os valores em ambos os *sets* em ordem classificada. O *set* de intersecção contém todos os valores que o *Set A* e *Set B* incluem, mas não os valores que um dos *sets* não inclui. Finalmente, o *set* de diferença contém os valores que aparecem no primeiro *set*, mas não no segundo. Quando você compilar e executar o programa, sua tela exibirá a seguinte saída:

```
Set A: aaa bbb ccc ddd eee fff
Set B: aaa ccc eee ggg hhh iii
União: aaa bbb ccc ddd eee fff ggg hhh iii
Interseção: aaa ccc eee
Set C (diferença de A e B): bbb ddd fff
C:\>
```

1250 UM EXEMPLO SIMPLES DE SET

Na dica anterior, você aprendeu sobre o objeto *set* e escreveu um programa simples, *set1.cpp*, que usava o tipo *set* e gerava saída na tela. Embora não haja espaço suficiente para explorar o tipo *set* em profundidade, esta dica apresenta as funções *lower_bound*, *upper_bound* e *equal_range* que seus programas podem usar com o tipo *set*.

A função *lower_bound* retorna um iterador para o elemento anterior na seqüência controlada que tem uma *key* que não é igual ao valor que o programa passou para a função *lower_bound*. A função *upper_bound* retorna um iterador para o elemento mais antigo na seqüência controlada que tem uma *key* que é igual ao valor que o programa passou para a função *upper_bound*. Se esse elemento não existe, a função retorna *end*. Em ambos os casos, o programa usa a função *set::key_comp(key, x)* para determinar se as *keys* são iguais. A função *equal_range* retorna um valor em par, onde *.first* é o resultado da função *lower_bound* e *.second* é o resultado da função *upper_bound*. Para compreender melhor o processamento que as funções-membro executam, considere o programa a seguir, *set_rang.cpp*:

```
#include <set.h>
#include <iostream.h>

using namespace std;
typedef set<int, less<int>> SET_INT;

void main(void)
 {
   SET_INT s1;
   SET_INT::iterator i;

   s1.insert(5);
   s1.insert(10);
   s1.insert(15);
   s1.insert(20);
   s1.insert(25);
   cout << "s1 -- iniciando em s1.lower_bound(12)" << endl;

// imprime: 15,20,25
   for (i = s1.lower_bound(12); i != s1.end(); i++)
      cout << "s1 tem " << *i << " em seu set." << endl;
   cout << "s1 -- iniciando em s1.lower_bound(15)" << endl;

// imprime: 15,20,25
   for (i = s1.lower_bound(15);i != s1.end(); i++)
      cout << "s1 tem " << *i << " em seu set." << endl;
   cout << "s1 -- iniciando em s1.upper_bound(12)" << endl;

// imprime: 15,20,25
```

```
    for (i = s1.upper_bound(12); i != s1.end(); i++)
       cout << "s1 tem " << *i << " em seu set." << endl;
    cout << "s1 -- iniciando em s1.upper_bound(15)" << endl;

// imprime: 20,25
    for (i = s1.upper_bound(15); i != s1.end(); i++)
      cout << "s1 tem " << *i << " em seu set." << endl;
    cout << "s1 -- s1.equal_range(12)" << endl;

// não imprime nada
    for (i = s1.equal_range(12).first;i != s1.equal_range(12).second; i++)
       cout << "s1 tem " << *i << " em seu set." << endl;
    cout << "s1 -- s1.equal_range(15)" << endl;

// imprime: 15
    for (i = s1.equal_range(15).first;i != s1.equal_range(15).second; i++)
       cout << "s1 tem " << *i << " em seu set." << endl;
 }
```

O programa *set_rang.cpp* ilustra como usar a função *lower_bound* para obter um iterador para o elemento mais antigo na seqüência controlada que tem uma key que não é igual ao valor passado para a função. Ele também ilustra como usar a função *upper_bound* para obter um iterador para o primeiro elemento na sequência controlada que tem uma *key* que é igual ao valor passado para a função A última coisa que ele ilustra é como usar a função *equal_range* para obter um valor par que contém os resultados *lower_bound* e *upper_bound* da *key*. Quando você compilar e executar o programa *set_rang.cpp*, sua tela exibirá a seguinte saída:

```
s1 -- iniciando em s1.lower_bound(12)
s1 tem 15 em seu set.
s1 tem 20 em seu set.
s1 tem 25 em seu set.
s1 -- iniciando em s1.lower_bound(15)
s1 tem 15 em seu set.
s1 tem 20 em seu set.
s1 tem 25 em seu set.
s1 -- iniciando em s1.upper_bound(12)
s1 tem 15 em seu set.
s1 tem 20 em seu set.
s1 tem 25 em seu set.
s1 -- iniciando em s1.upper_bound(15)
s1 tem 20 em seu set.
s1 tem 25 em seu set.
s1 -- s1.equal_range(12)
s1 -- s1.equal_range(15)
s1 tem 15 em seu set.
```

INTRODUZINDO A PROGRAMAÇÃO EM WIN32 1251

Nas 1250 dicas anteriores, você aprendeu como programar em C e C++. Embora os programas escritos nas dicas anteriores rodassem sob o Windows, eles foram projetados mais para um ambiente DOS ou Unix. Nas próximas 250 dicas, você aprenderá os fundamentos da programação em Windows. Muitas dicas que virão, revisitarão as dicas anteriores com explicações de como executar processamento similar usando a Interface de Programação de Aplicativos do Windows, que este livro e a maioria dos outros livros sobre a programação W.ndows referenciam como *Win32 API*. É importante compreender que este livro enfoca a Win32 API, não a Win16 API — o que significa que as chamadas de funções que este livro discute funcionarão sob o Windows 95 e sob o Windows NT, mas podem não funcionar sob o Windows 3.11 ou versões anteriores.

Adicionalmente, por causa das diferenças nos dois sistemas operacionais, haverão casos em que uma dica introduzirá uma função da API que o Windows 95 ou o Windows NT, mas não ambos, suportam. Tipicamente, essas dicas notarão qual sistema operacional não suporta a função da API e por que ela não suporta. Se você tiver

problemas com o programa que uma determinada dica contém, poderá querer reverificar a dica para garantir que o código é compatível com seu sistema operacional.

Finalmente, como você talvez saiba, a saída que o Windows gera é muito diferente da saída que você produziu até aqui neste livro. Na verdade, um único programa do DOS que exibe "Bíblia do Programador C/C++" gera saída, como segue:

```
Bíblia do Programador C/C++
C:\>
```

Por outro lado, um programa Windows simples, que cria uma caixa de mensagem que contém informações similares geraria a saída mostrada na Figura 1251.

Figura 1251 Uma caixa de mensagem simples do Windows.

1252 Mais Diferenças entre os Programas Windows e DOS

Como você viu na dica anterior, existem diferenças significativas na aparência da saída que seus programas irão gerar sob o Windows e a aparência da saída que seus programas DOS geraram. No entanto, as diferenças entre a programação para o DOS e para o Windows são muito mais significativas do que apenas a apresentação, embora essa seja uma das diferenças mais facilmente compreendidas. A lista a seguir detalha algumas das outras diferenças importantes entre a programação para o Windows e para o DOS:

- Como o Windows é um sistema operacional multitarefa (em que dois ou mais programas podem rodar ao mesmo tempo), seus programas estarão compartilhando espaço na memória com pelo menos um programa, e frequentemente muitos outros programas. É muito importante certificar-se de que seu programa permanecerá dentro de seu espaço alocado na memória, e não modificará o espaço de memória de outros programas.
- Como o Windows processa informações baseadas em texto de forma diferente da forma do DOS, você não mais gerará saída usando *printf* ou *cout*. No entanto, você frequentemente poderá usar matrizes de canais para formatar informações antes de enviá-las para uma janela.
- Como a interface gráfica do Windows pode receber entrada do usuário em virtualmente qualquer ponto no processamento de um programa, em geral seus programas Windows não serão tão lineares quanto seus programas do DOS. Em outras palavras, você criará seus programas Windows que respondem às ações do usuário e às mensagens do sistema.
- Seus programas Windows geralmente consistirão de múltiplas classes em múltiplos arquivos de classe e de cabeçalho, e, normalmente, serão, de forma significativa maiores do que os programas DOS porque não estão limitados à memória convencional.
- Os programas Windows que você projetar neste livro são geralmente mais capazes de receber entrada do usuário que muitos dos programas DOS que você criou em dicas anteriores. Como regra, como a natureza gráfica do Windows requer interação do usuário para processamento útil, a maioria dos programas que você escrever para o Windows suportará algum tipo de interação do usuário.
- A despeito de todas as diferenças que esta lista discute, e todas as diferenças que você aprenderá em dicas posteriores, é importante lembrar que a programação em C++ para o Windows é fundamentalmente a mesma que a programação C++ que você fez até aqui — você apenas aplicará muitos desses conceitos de um modo ligeiramente diferente do que faz com seus programas DOS.

Nas próximas dicas, você aprenderá alguns fundamentos da programação para o Windows, e o impacto que esses fundamentos terão em todos os programas Windows. Você então começará a escrever programas Windows e, ao tempo em que completar este livro, poderá escrever programas Windows que manipulam memória, arquivos, gráficos, impressoras e mais.

Nota: *O compilador Turbo C++ Lite incluído no CD-ROM que acompanha este livro não tem a capacidade de gerar código para o Windows. Para escrever programas Windows, você precisará ter um compilador completo, tal como o* **Visual C++** *ou o Borland* **C++ 5.02**. *Os programas apresentados nas dicas a seguir trabalharão com um desses compiladores.*

INTRODUZINDO OS ENCADEAMENTOS 1253

Como você aprendeu na seção DOS deste livro, como uma regra geral, o DOS suporta a execução de apenas um programa de cada vez dentro da memória. Como você também aprendeu, o Windows não está sujeito a essa limitação. Na verdade, ele limita o número de programas que um computador pode rodar de cada vez somente se não houver memória suficiente para abrir programas adicionais. O Windows gerencia todos esses programas na memória usando *encadeamentos*. Basicamente, um encadeamento é um modo de você poder visualizar a solicitação de um programa para usar a CPU do computador. Quando você tiver múltiplos programas em execução simultânea, o sistema operacional gerenciará um ou mais encadeamentos para cada um desses programas. O Windows coloca cada encadeamento em uma fila de encadeamentos (uma lista). Mais tarde, dependendo da prioridade dos encadeamentos, o Windows pegará um encadeamento da fila e o atribuirá à CPU, a qual, por sua vez, executará as instruções do encadeamento. A prioridade de um encadeamento determina se o Windows coloca o encadeamento na frente ou após outros encadeamentos dentro da fila. A Figura 1253.1 mostra uma representação gráfica do esquema de multitarefa com encadeamentos do Windows.

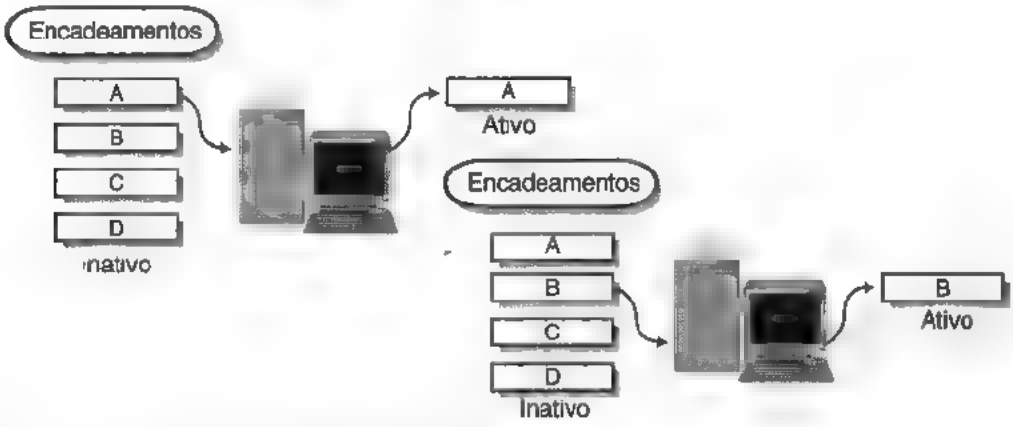

Figura 1253.1 A CPU processa uma série de encadeamentos.

Você aprenderá sobre os encadeamentos em detalhes em dicas posteriores. No entanto, por ora, compreenda que um encadeamento é a entidade na qual o Windows faz a CPU rodar os comandos do programa. O Windows classifica os encadeamentos dependendo da prioridade deles, e a CPU do computador executa cada encadeamento sequencialmente. A Figura 1253.2 mostra um modelo simples de como seu programa solicita processamento da CPU e recebe suas informações quando a CPU completa o processamento.

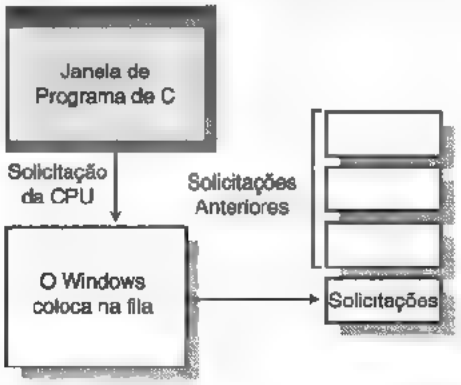

Figura 1253.2 Um modelo simples do esquema de processamento de encadeamentos do Windows.

Como você aprenderá na dica a seguir, o Windows retorna informações para seus programas toda vez que *processa um encadeamento para esse programa na forma de uma mensagem.*

1254 COMPREENDENDO AS MENSAGENS

O modo fundamental de comunicação que o Windows e os programas escritos para o Windows usam são as *mensagens.* Basicamente, toda vez que uma operação ocorre, o Windows responde a essa operação ou ação enviando uma mensagem para si próprio ou para outro programa. Por exemplo, quando um usuário dá um clique no mouse dentro da janela do seu programa, o Windows lê esse clique e envia uma mensagem para o seu programa avisando que o usuário deu um clique no mouse em uma determinada posição dentro da janela do programa. Seu programa, ao receber essa mensagem, começará seu próprio processamento conforme for apropriado para a mensagem. Se a mensagem nao for importante para o programa, ele deverá simplesmente ignorá-la. Para compreender melhor o modelo de mensagem do Windows, considere a Figura 1254, que mostra o modelo em uma forma linear e simplificada.

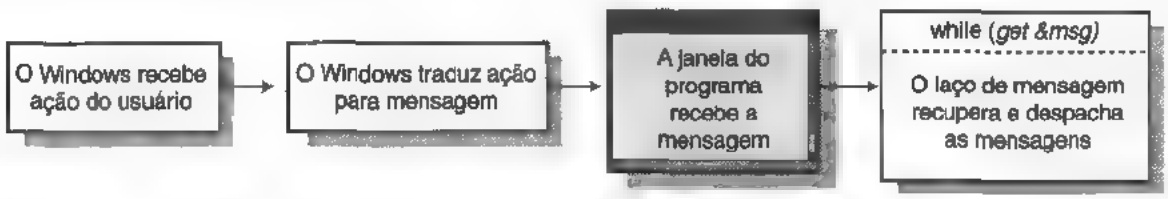

Figura 1254 Um diagrama simples do modelo de mensagens do Windows.

Quando você escrever programas para o Windows, as rotinas mais importantes dentro de seus programas serão aquelas que aceitam e processam mensagens do sistema operacional. Por exemplo, o fragmento de código a seguir da função *ProcJan* exibe um exemplo de processamento de mensagem e função de resposta:

```
LRESULT CALLBACK ProcJan(HWND hJan, UNIT uMsg, WPARAM wParam, LPARAM lParam)
 {
   switch(uMsg)
   {
      case WM_COMMAND:
         switch(LOWORD(wParam))
         {
            case IDM_TEST :
               break;
            case IDM_EXIT :
               DestroyWindow(hJan);
               break;
         }
      break;
      case WM_DESTROY :
         PostQuitMessage(0);
         break;
      default:
         return (DefWindowProc(hJan, UMsg, wParam, lParam));
   }
   return(0L);
 }
```

Em geral, o fragmento de código usa um comando *switch* para determinar o tamanho da mensagem. À medida que você estudar as dicas a seguir, rapidamente aprenderá e compreenderá o processamento executado dentro da função *ProcJan.* No fragmento anterior, a função verifica a ocorrência de uma mensagem *WM_COMMAND* e *WM_DESTROY,* ou passa a função para a rotina de tratamento de mensagem padrão. Com base em qual mensagem recebe, a função efetua o processamento apropriado.

COMPREENDENDO OS COMPONENTES DE UMA JANELA 1255

Como você deve ter imaginado, o bloco de construção para todos os programas Windows é uma ou mais janelas ou caixas de diálogo (um tipo especial de janela). Na verdade, quase todo objeto dentro de uma janela é também uma janela de algum tipo. Em dicas posteriores, você compreenderá melhor como as janelas que seus programas usarão, serão similares, mas derivadas de posições diferentes Para esta dica em particular, é importante compreender os componentes de uma "janela padrão" — em outras palavras, que seus usuários considerarão uma janela.

O Windows geralmente compõe cada "janela padrão" a partir de sete partes básicas. Você pode dividir ainda mais cada uma dessas partes, o que fará em dicas posteriores. No entanto, é importante compreender o quadro maior da janela antes de analisar cada uma das partes pequenas que compõem um único componente de janela. Os sete componentes básicos de uma janela são mostrados na lista a seguir:

- A *moldura de janela* é o repositório para tudo o mais dentro de uma janela. Como você aprenderá, pode criar molduras de janelas de muitos tipos diferentes. O tipo mais comum de moldura de janela é uma janela de tamanho redefinível, similar àquela mostrada na Figura 1255.1. Dentro de seus programas, você manipulará a moldura e receberá muitas mensagens (tais como mensagens de redefinição do tamanho da janela) da moldura. Dicas posteriores discutirão a moldura da janela em detalhes.
- A *barra de título* fornece informações para o usuário sobre o programa. A barra de título estende a largura da janela na borda superior. Ela identifica o que a janela mostra e também permite que o usuário execute muitas operações de janelas A barra de título é o ponto de controle para mover a janela e as localizações do menu do sistema: os botões *Maximizar, Minimizar, Restaurar* e *Fechar Janela*. Como você aprenderá em dicas posteriores, a composição da barra de título variará bastante dependendo da aplicação que cria a janela e do propósito da janela dentro da qual a barra de título está.
- Os botões *Minimizar, Maximizar* e *Fechar Janela*, embora dentro do repositório da barra de título, são importantes o suficiente para você os considerar como componentes da própria janela. Os botões lhe permitem controlar o tamanho de uma janela e fechar a janela quando você terminar de usá-la.
- A *área cliente* é a área dentro da janela que seus programas podem personalizar e que seus programas devem projetar com a intenção de receber entrada do usuário. Em outras palavras, a área cliente é a seção da janela onde ocorre a maior parte da ação. Por exemplo, se você estiver trabalhando em um documento do Microsoft *Word*®, a área cliente é a região onde você digita e edita seu documento dentro da janela.
- As *barras de rolagem* permitem que o usuário navegue para a esquerda e para a direita, para cima e para baixo dentro de uma janela. Por exemplo, seus programas poderiam usar as barras de rolagem, permitindo que o usuário veja mais de um formulário de entrada. Seus programas também poderiam usar as barras de rolagem, permitindo que o usuário percorra um documento que salvou anteriormente no disco. As barras de rolagem são uma ferramenta navegável importante dentro dos programas Windows. O equivalente mais próximo dentro de um programa DOS poderiam ser as teclas de setas, dependendo de como você implementou as teclas dentro de um determinado programa.
- A *barra de menu* é um componente da maioria das janelas-mãe, mas geralmente não está presente dentro da maioria das janelas-filha. (A dica a seguir explica as janelas-mãe e as janelas-filha detalhadamente.) Os programas Windows usam a barra de menu para fornecer ao usuário opções apropriadas para o programa. Como você aprenderá, a maioria dos programas Windows inclui, no mínimo, um menu Arquivo e um menu Ajuda. Os programas Windows mais complexos podem incluir 10 ou mais menus, e cada menu pode conter 20 ou mais seleções. À medida que seus programas se tornarem mais complexos, seus menus se tornarão mais complexos. A Figura 1255.1 mostra a barra de menu do Microsoft *Word*.

Figura 1255.1 A barra de menu do Microsoft *Word*.

- A *barra de status* é um componente da maioria das janelas-mãe, mas geralmente não está presente dentro da maioria das janelas-filha. Os programas Windows usam a barra de status para fornecer ao usuário informações específicas que detalham a posição atual do usuário dentro do programa — seja uma posição dentro de um documento, uma linha e assim por diante. A Figura 1255.2 mostra a barra de status do Microsoft *Word*, que fornece informações importantes sobre a posição atual do usuário dentro do documento atual.

Figura 1255.2 A barra de status do Microsoft *Word*.

1256 COMPREENDENDO AS JANELAS-MÃE E AS JANELAS-FILHA

Na dica anterior, você aprendeu que a maioria das janelas-mãe inclui a barra de menu e a barra de status, enquanto que a maioria das janelas-filha não inclui. No entanto, talvez você ainda não tenha compreendido o que são as janelas-mãe e as janelas-filha. A maioria dos programas Windows suporta a Interface de Múltiplos Documentos (MDI — Multiple Document Interface), o que permite que um único programa mantenha e exiba múltiplos componentes dentro de uma única janela. Por exemplo, a Figura 1256.1 mostra o Microsoft *Word* com quatro janelas de documento abertas.

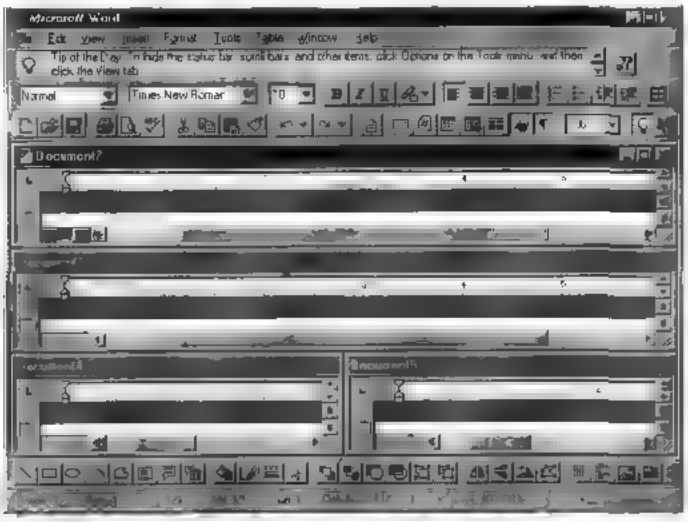

Figura 1256.1 A janela-mãe do Microsoft *Word* com quatro janelas-filha abertas.

De forma muito parecida com a hierarquia de classes que você aprendeu e usou dentro de vários programas C++, toda janela no Windows deriva de uma janela básica. Assim, toda janela tem uma janela-mãe. Na Figura 1256.1, as janelas-filha são as janelas de documento internas, e a janela-mãe é a janela do Microsoft *Word*. No entanto, você também poderia considerar a janela do Microsoft *Word* como uma filha; e a Área de Trabalho do Windows como sua janela-mãe. A Área de Trabalho do Windows não tem uma janela-mãe.

Na figura anterior, a janela-mãe suporta a interface de múltiplos documentos, o que permite que ela tenha muitas janelas-filha. Outros programas Windows suportam a interface de múltiplos documentos, o que permite que a janela-mãe tenha somente uma janela-filha, como mostrado na Figura 1256.2.

Finalmente, muitos programas Windows mais novos suportam uma variação especial da interface de um único documento, que é comumente conhecida como a *interface de documento no estilo do Explorer*. Os programadores chamam a variante de interface de documento no estilo do Explorer porque eles modelam a interface com base no programa Windows Explorer. A interface de documento é similar à interface de documento único em todos os sentidos, exceto a janela de um único documento que está dividida ao meio, facilitando para o programa exibir conjuntos de dados únicos dentro de uma única exibição. A Figura 1256.3 mostra o Windows Explorer e a interface no estilo do Explorer

Figura 1256.2 O *Bloco de Notas* do Windows é um programa de um único documento.

Figura 1256.3 O Windows *Explorer* e a interface no estilo do Explorer.

Dentro de seus programas, você freqüentemente manipulará muitas janelas-filha dentro de uma única janela-mãe. À medida que você for trabalhando com o projeto de janelas em dicas posteriores, aprenderá mais sobre as importantes diferenças entre os vários estilos de interface.

CRIANDO UM PROGRAMA WINDOWS GENÉRICO 1257

Como você descobrirá, quase todos os programas Windows executam uma certa quantidade de processamento padrão (sobrecarga) necessária para o programa rodar. Basicamente, o programa precisa criar pelo menos uma janela, registrar essa janela com o sistema operacional e tratar as mensagens enviadas do sistema operacional para a janela. Adicionalmente a maioria dos programas Windows usa um arquivo adicional chamado *arquivo de recurso*. Os arquivos de recurso informam ao compilador as características da janela que o programa cria. A dica a seguir explicará os arquivos de recursos em detalhes. Após você criar a forma básica do programa Windows, seus outros programas geralmente usarão essa forma básica como a pedra angular para a criação do programa. Para minimizar a possível confusão com programas posteriores, a forma básica do programa é chamada de *generico.cpp*. O CD-ROM que acompanha este livro inclui todas as dependências (arquivos componentes) para *generico.cpp*. No entanto, como dicas posteriores explicarão essas dependências em detalhes, esta dica somente apresentará *generico.cpp*, como mostrado aqui:

```
#include <windows.h>
#include "generic.h"

HINSTANCE hCop;         // ocorrência atual
LPCTSTR lpszNomeAplic = "Genérico";
LPCTSTR lpszTitulo =  "Aplicativo Genérico";
BOOL RegisterWin95(CONST WNDCLASS* lpwc);

int APIENTRY WinMain(HINSTANCE hCopia, HINSTANCE hCopiaAnt,
                     LPSTR lpLinhaCmd,
```

```
                                int nCmdExibe)
 {
   MSG msg;
   HJAN hJan;
   WNDCLASS cj;

   cj.style          = CS_HREDRAW | CS_VREDRAW;
   cj.lpfnWndProc    = (WNDPROC)ProcJan;
   cj.cbClsExtra     = 0;
   cj.cbWndExtra     = 0;
   cj.hInstance      = 0;
   cj.hIcon          = LoadIcon(hCopia, lpszNomeAplic);
   cj.hCursor        = LoadCursor(NULL, IDC_ARROW);
   cj.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
   cj.lpszMenuName   = lpszNomeAplic;
   cj.lpszClassName  = lpszNomeAplic;

   if(!RegisterWin95(&cj))
      return false;
   hCop = hCopia;
   hJan = CreateWindow (lpszNomeAplic, lpszTitulo,
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                        CW_USEDEFAULT, 0,
                        NULL, NULL, hCopia, NULL);
   if(!hJan)
      return false;
   ShowWindow(hJan, nCmdExibe);
   UpdateWindow(hJan);
   while(GetMessage(&msg, NULL, 0,0))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
   return(msg.wParam);
 }

BOOL RegisterWin95(CONST WNDCLASS* lpcj)
 {
   WNDCLASSEX cjex;

   cjex.style          = lpcj->style;
   cjex.lpfnWndProc    = lpcj->lpfnWndProc;
   cjex.cbClsExtra     = lpcj->cbClsExtra;
   cjex.cbWndExtra     = lpcj->cbWndExtra;
   cjex.hInstance      = lpcj >hCopia;
   cjex.hIcon          = lpcj->hIcon;
   cjex.hCursor        = lpcj->hCursor;
   cjex.hbrBackground  = lpcj->hbrBackground;
   cjex.lpszMenuName   = lpcj->lpszMenuName;
   cjex.lpszClassName  = lpcj->lpszClassName;
   cjex.cbSize         = sizeof(WNDCLASSEX);
   cjex.hIconSm        = LoadIcon(cjex.hCopia, "PEQUENO");
   return RegisterClassEx(&cjex);
 }

LRESULT CALLBACK ProcJan(HJAN hJan, UINT uMsg, WPARAM wParam, LPARAM lParam)
 {
   switch(uMsg)
    {
      case WM_COMMAND:
         switch(LOWORD(wParam))
```

```
            {
              case IDM_TESTAR :
                 break;
              case IDM_SAIR :
                 DestroyWindow(hJan);
                 break;
            }
           break;
        case WM_DESTROY :
           PostQuitMessage(0);
           break;
        default:
           return (DefWindowProc(hJan, uMsg, wParam, lParam));
     }
   return(0L);
}
```

Como você pode ver, o programa *generico.cpp* consiste de três funções: *WinMain* (o equivalente de *main* para os programas Windows), *RegisterWin95* e *ProcJan*. Como você viu na Dica 1254, a função *ProcJan* trata as mensagens que o sistema operacional envia para o programa. A função *RegisterWin95* trata alguma limpeza extra (especificamente, acrescenta mais informações ao objeto *WNDCLASS*, que o Windows 95 requer mas que o Windows NT não requer) e registra a nova janela com o sistema operacional. Você aprenderá mais sobre o registro de janelas em dicas posteriores. Quando você compilar e executar o programa *generico.cpp*, sua tela exibirá a saída mostrada na Figura 1257.

Figura 1257 A saída do programa *generico.cpp*.

COMPREENDENDO OS ARQUIVOS DE RECURSOS 1258

Como você aprendeu na Dica 1255, até mesmo os programas Windows mais simples têm muitos componentes. No entanto, muitos desses componentes são relativamente estáticos — o que significa que seus programas não os alterarão com frequência. Os menus, por exemplo, raramente mudam dentro da maioria dos programas. Você também raramente alterará os ícones, as informações da barra de título e assim por diante dentro dos programas, se alterar. Adicionalmente, o Windows espera que seus programas armazenem informações sobre si próprios (autor, número da versão etc.) junto com o próprio programa. O Windows usa essas informações para fornecer ao usuário informações sobre o programa. Por exemplo, se você selecionar o programa *generico.exe* dentro do Windows *Explorer* e pressionar as teclas de barra de espaço, o Windows exibirá a caixa de diálogo Propriedades do Arquivo. Além de exibir informações básicas sobre o arquivo, você pode dar um clique com seu mouse na guia *Versão*, o que faz a caixa de diálogo Propriedades do Arquivo aparecer, como mostrado na Figura 1258.

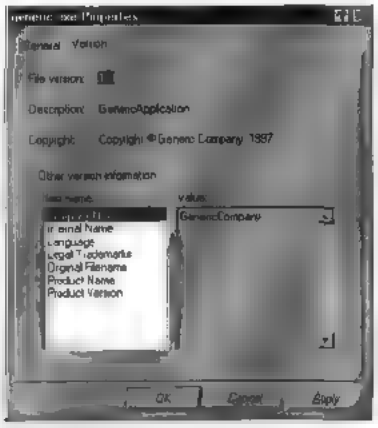

Figura 1258 A caixa de diálogo propriedades do arquivo para o programa *generico.cpp*.

Como você pode ver, o Windows armazena informações sobre o arquivo dentro do atributo Propriedades. Na verdade, o criador do programa atribuiu essas propriedades ao programa *generico.exe* dentro do arquivo de recursos *generico.rc* (uma das dependências mencionadas na dica anterior). O arquivo de recursos *generico.rc* é um arquivo de texto que mantém informações sobre muitos dos recursos estáticos que o programa usará. O arquivo de recursos *generico.rc* contém informações sobre os recursos que o programa usará após a execução, incluindo menus, ícones e assim por diante. O arquivo *generico.rc* contém listagens, definições de menus e informações sobre a propriedade do arquivo, como mostrado aqui:

```
#include "windows.h"
#include "generic.h"
#include "winver.h"

MEUAPLIC ICON  DISCARDABLE 'GENERICO.ICO'
PEQUENO  ICON  DISCARDABLE 'PEQUENO.ICO"

GENERICO MENU DISCARDABLE
BEGIN
   POPUP "&Arquivo"
   BEGIN
      MENUITEM "&Testar!",              IDM_TESTAR
      MENUITEM "&Sair",                 IDM_SAIR
   END
   POPUP "&Ajuda"
   BEGIN
      MENUITEM "&Sobre MeuAplic...",    IDM_SOBRE
   END
END

 VERSIONINFO
 FILEVERSION 3,3,0,0
 PRODUCTVERSION 3,3,0,0
 FILEFLAGSMASK 0X3f1
#ifdef _DEBUG
   FILEFLAGS 0xb1
#else
   FILEFLAGS 0xa1
#endif
 FILEOS 0X4L
 FILETYPE 0X1L
 FILESUBTYPE 0X0L
BEGIN
   BLOCK "StringFileInfo"
   BEGIN
      BLOCK "040904B0"
```

```
        BEGIN
            VALUE "CompanyName", "GenericCompany\0"
            VALUE "FileDescription", "GenericApplication\0"
            VALUE "FileVersion", "1.0\0"
            VALUE "InternalName", "1.0\0"
            VALUE "LegalCopyright", "Copyright \251 Generic Company. 1997\0"
            VALUE "LegalTrademarks", "Generic Trademark.\0"
            VALUE "OriginalFilename", "\0"
            VALUE "ProductName", "Generic Application.\0"
            VALUE "ProductVersion", "1.0\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END
```

O arquivo de recursos *generico.rc* é um arquivo de recursos simples. À medida que seus programas se tornarem mais complexos, você verá que projetará arquivos de recursos com centenas e até milhares de linhas. O arquivo de recursos *generico.rc* realmente trata apenas dois tipos de recursos: armazena as informações para a caixa de diálogo Propriedades de Arquivo, como você viu anteriormente nesta dica, e armazena as informações a partir das quais cria o menu para o programa. Quando você usar o compilador de recursos para compilar o arquivo de recursos *generico rc*, o compilador usará as informações dentro do arquivo para gerar as informações dos menus e propriedades para o arquivo de programa executável. No arquivo de recursos *generico.rc*, o menu consiste de dois menus suspensos (Arquivo e Ajuda), cada um dos quais tem um ou mais itens.

É importante compreender que os programas não têm que usar as informações que o arquivo de recursos oferece. Por exemplo, o programa *generico.cpp* suporta uma barra de menu porque criou a janela com um ponteiro para a barra de menu *generica* no arquivo *generico.rc*. Implementar um arquivo de recurso não é suficiente; você precisa implementar os recursos dentro de seus programas de algum modo.

Nota: Muitas ferramentas de desenvolvimento em C++ para o Windows incluem a capacidade de arrastar-e-soltar, que você pode usar para evitar a criação de arquivos de recurso de texto. Com as ferramentas de arrastar-e-soltar, você pode acrescentar itens diretamente nos menus ou em uma janela sem criar manualmente as informações do arquivo de recurso para esses recursos. No entanto, o projeto de arrastar-e-soltar é único para cada compilador. Como virtualmente não há portabilidade entre os compiladores, o arquivo de recursos é útil para garantir que os programas neste livro sejam consistentes de leitor para leitor.

COMPREENDENDO OS INDICATIVOS DO WINDOWS 1259

Como você aprendeu na seção Arquivos e E/S anteriormente, quando seus programas trabalham com arquivos e discos, eles podem fazer isso em baixo nível, com as rotinas da BIOS, ou em alto nível, com indicativos (também chamados de *handles*) de arquivos. Dentro do Windows, você também usará indicativos para manter informações sobre os arquivos. No entanto, você usará um tipo diferente de indicativo, um *indicativo de janela*, para obter ou manter informações sobre uma janela dentro de seu programa ou em alguma outra parte dentro no sistema. Um indicativo de janela é basicamente um valor *long* que você mantém dentro de uma variável do tipo *HWND*. Quando você chamar uma função da API que requeira um indicativo de arquivo, passe a variável do tipo *HWND* para a função. O Windows, por sua vez, verificará o indicativo da janela em sua lista de indicativos válidos, e, então, enviará a mensagem para, ou executará a ação na janela apropriada. A Figura 1259 mostra um modelo lógico do processo de avaliação do indicativo da janela do Windows.

Dentro de seus programas, você usará indicativos para janelas para lhe ajudar a controlar a aparência, o tamanho e outras qualidades de uma janela.

Como você aprenderá em dicas posteriores, C++ para o Windows define um grande número adicional de tipos, dos quais *HWND* é somente um. Por exemplo, o programa *generico.cpp* que você escreveu na Dica 1257 inclui seis novos tipos, que seus programas não-Windows não usarão. Dicas posteriores examinarão os novos tipos em detalhes.

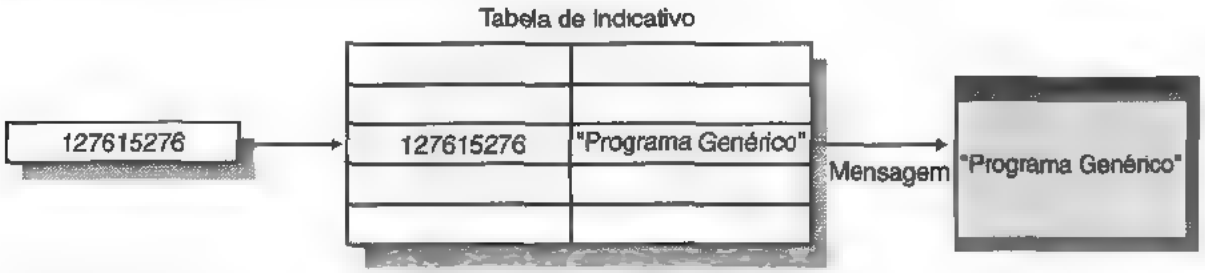

Figura 1259 O processo de avaliação do indicativo da janela do Windows.

1260 DEFININDO OS TIPOS DE INDICATIVO DO WINDOWS

Na dica anterior você aprendeu sobre o tipo *HWND*, que usará dentro de seus programas Windows para manter um valor *long* que representa uma janela aberta. O Windows na realidade define doze tipos de indicativos. Por exemplo, o programa *generico.cpp* que você criou na Dica 1257 incluiu uma definição de uma variável do tipo *HWND*, e também incluiu uma definição de uma variável do tipo *HINSTANCE*. Como o Windows é um sistema operacional de multitarefa, é possível ter múltiplas cópias, ou *ocorrências*, de um programa em execução ao mesmo tempo. O Windows mantém um número único para cada ocorrência, que armazena dentro de uma tabela de indicativos do tipo *HINSTANCE*. Seus programas podem usar uma variável *HINSTANCE* para manter informações sobre qual ocorrência está atualmente em execução. Em alguns casos, você pode usar um valor *HINSTANCE* junto com uma chamada da API do Windows para determinar quantas ocorrências atuais de um programa estão em execução. Na dicas restantes deste livro, você encontrará diferentes tipos de indicativos. Dicas posteriores explicarão cada tipo de indicativo à medida que você os encontrar dentro de seus programas. A Tabela 1260 lista os tipos de indicativo do Windows.

Tabela 1260 Tipos de Indicativo do Windows.

Tipo de Indicativo	Descrição
HANDLE	Número que identifica de forma única um indicativo
HBITMAP	Número que identifica de forma única um mapa de bits (*bitmap*)
HBRUSH	Número que identifica de forma única um pincel
HCURSOR	Número que identifica de forma única um cursor
HFONT	Número que identifica de forma única uma fonte
HGDIOBJ	Número que identifica de forma única um objeto da interface de dispositivo gráfico (GDI)
HICON	Número que identifica de forma única um ícone
HINSTANCE	Número que identifica de forma única uma ocorrência
HPALETTE	Número que identifica de forma única uma paleta
HPEN	Número que identifica de forma única uma caneta
HRGN	Número que identifica de forma única uma região
HWND	Número que identifica de forma única uma janela

É importante reconhecer que seus programas acessarão a maioria dos tipos de indicativos para controlar a exibição de informações dentro das janelas. C++ suporta indicativos de arquivo para o Windows suficientemente, de modo que o Windows necessita de novos tipos de indicativo apenas para ajudá-lo (e ajudar você) a controlar a exibição.

1261 COMPREENDENDO O ARQUIVO DE CABEÇALHO GENÉRICO

Antes de começar a analisar os componentes do programa *generico.cpp* mais de perto, é importante reconhecer a partir do código original que o programa requer o uso do arquivo *generico.h*, como mostrado aqui:

```
#define IDM_SAIR        100
#define IDM_TESTAR      200
#define IDM_SOBRE       300
LRESULT CALLBACK ProcJan(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK Sobre  (HWND, UINT, WPARAM, LPARAM);
```

O arquivo de cabeçalho *generico.h* define diversas constantes. Você deve reconhecer essas constantes a partir da declaração de menu dentro do arquivo de recursos — os nomes das constantes correspondem aos identificadores dentro do arquivo de recursos. Dicas posteriores usarão as constantes para identificar quando o usuário selecionou um item de menu dentro do arquivo. As duas últimas linhas definem os protótipos para as funções *ProcJan* e *Sobre*. Você criou a função *ProcJan* na Dica 1257, e criará a função *Sobre* posteriormente. No entanto, observe que o arquivo de cabeçalho define os protótipos para ambas as funções com *callback*. A Dica 1262 discute as funções *callback* em detalhes.

COMPREENDENDO AS FUNÇÕES CALLBACK 1262

Na dica anterior e na Dica 1257, você viu diversas funções declaradas com a palavra-chave *CALLBACK*. Referenciaremos essas funções como *funções de callback*. Uma função de *callback* é uma função para a qual você passa o endereço de uma terceira função, que essa terceira função então "chama de volta" com informações. Você sempre definirá a função *ProcJan* como uma função de *callback*. Dentro de seus programas, você normalmente usará as funções de *callback* com funções específicas da API, tais como *EnumFontFamilies* e *EnumWindows*. Quando você passar o endereço de uma função de *callback* para uma dessas funções, a função chamará a função de *callback* para cada item dentro da lista. Por exemplo, se você chamar *EnumWindows*, provavelmente passará para ela o endereço de uma função de *callback* que exibe os valores ou os acrescenta a uma matriz. *EnumWindows*, por sua vez, chamará a função de *callback* para cada janela dentro de sua lista de todas as janelas.

As funções de *callback* são necessárias no Windows porque seus programas tratarão muitas de suas ações importantes por meio da Interface de Programação de Aplicativos do Windows (API) — que seus programas não podem modificar diretamente. Portanto, você precisará fornecer a API os meios de poder chamar suas rotinas personalizadas quando retornar informações estendidas (tais como uma lista). A dica a seguir detalha mais a API do Windows.

APRESENTANDO A INTERFACE DE PROGRAMAÇÃO DE APLICATIVOS DO WINDOWS 1263

Como você aprendeu, a Interface de Programação de Aplicativos (API) do Windows é o alicerce da maioria dos programas no Windows. Em geral, qualquer ação que seu programa execute no Windows além das operações matemáticas mais simples, usam a API do Windows, direta ou indiretamente (em outras palavras, até definir um item de menu dentro de um arquivo de recurso usa a API do Windows, embora seu programa não chame as funções dentro da API diretamente). *No entanto, você verá que seus programas com freqüência chamam funções* dentro da API diretamente, como mostrado aqui:

```
hJan = CreateWindow (lpszNomeAplic, lpszTitulo,
                     WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                     0, CW_USEDEFAULT, 0, NULL, NULL,
                     bCopia, NULL);
```

CreateWindow é uma função da API do Windows que quase todo programa baseado em Windows usará para criar janelas. Como pode ver, você chamará a função (e a maioria das outras funções da API Win32) dentro de seus programas. Seus programas podem chamar as funções da API como se eles definissem as funções por uma única razão: todos os programas Windows que você escrever incluem o arquivo de cabeçalho *windows.h*. Esse arquivo de cabeçalho, por sua vez, inclui vários outros arquivos de cabeçalho, principalmente *winbase.h*, que contém as definições para as funções da API do Windows, bem como várias estruturas e tipos enumerados (tais como *HWND*) que seus programas usarão.

Infelizmente, a API do Windows é grande demais e não é possível listar todas as suas funções aqui existem mais de 1.500 funções básicas e mais de 2.000 outras que o software específico para o sistema operacional

(tal como o novo *Internet Explorer® 4.0*) acrescenta à API. A Microsoft estima que o novo Windows ® 98 conterá aproximadamente 4.000 funções da API.

1264 EXAMINANDO MELHOR O PROGRAMA GENERICO.CPP

Na Dica 1257, você criou o programa *generico.cpp*, que implementa os requisitos básicos para um programa Windows. Antes de começar a aprender mais sobre a programação Windows e começar a manipular funções mais complexas, é útil compreender exatamente quais ações seu programa *generico.cpp* executou. Nas próximas dicas, você analisará atentamente a função *WinMain*, o processo de criação de janelas e mais. No entanto, você também deve compreender o significado das variáveis globais que o programa *generico.cpp* fornece, como mostrado aqui:

```
#if defined (win32)
   #define IS_WIN32 TRUE
#else
   #define IS_WIN32 FALSE
#endif

HINSTANCE  hCop;            // ocorrência atual
LPCTSTR  lpszNomeApl = "Genérico";
LPCTSTR  lpszTitulo = "Aplicativo Genérico";
BOOL RegisterWin95(CONST WNDCLASS *lpcj);
```

Primeiro, o programa verifica a constante *win32* para determinar se o compilador está ou não efetuando uma compilação Win32. O programa verifica a compilação Win32 por várias razões, mas principalmente porque, como você sabe, existem diferenças significativas entre as APIs Win32 e Win16. Um programa que você projeta para rodar no Windows 3.11, mas escreve em uma máquina equipada com o Windows 95, precisa se limitar somente às chamadas da API dentro da API Win16, e seus programas podem usar a constante *IS_WIN32* para controlar quais funções da API seus programas chamam.

Em seguida, o programa define a variável *hCop*. Como você aprendeu na Dica 1260, *HINSTANCE* é um indicativo do Windows que mantém um número único que corresponde à ocorrência de execução atual do programa e nenhuma ocorrência desse ou de qualquer outro programa.

As declarações então definem *lpszNomeApl* e *lpszTitulo*, que parecem, à primeira vista, ser matrizes de caracteres, ou, talvez, até variáveis string. Você usará o tipo *LPCTSTR* (um tipo definido pelo Windows) para conter ponteiros string de leitura somente. Quando o compilador compilar seu programa, ele realmente converterá todas as declarações *LPCTSTR* para *const char FAR**. No entanto, como você pode ver, *LPCTSTR* é claramente mais fácil de digitar e de compreender que *const char FAR**.

Finalmente, o programa cria uma declaração de protótipo para a função *RegisterWin95*. A função *RegisterWin95* aceita um parâmetro do tipo *WNDCLASS* e retorna um valor booleano de sucesso. Você aprenderá mais sobre o registro do Windows na Dica 1269. No entanto, por enquanto, deve compreender que o tipo *WNDCLASS* contém informações que o Windows usa cada vez que registra ou cria uma nova janela (tal como a legenda na barra de título, o tipo da moldura e assim por diante).

1265 COMPREENDENDO A FUNÇÃO WINMAIN

No programa *generico.cpp*, que você criou na Dica 1257, a primeira função do programa foi *WinMain*. Como você aprendeu, a função *WinMain* é equivalente, em Windows, à função *main* que todos os seus programas C e C++ usaram para seu processamento principal. No entanto, a função *WinMain* difere de vários modos diferentes, como mostrado aqui:

```
int APIENTRY WinMain (HINSTANCE hCopia, HINSTANCE hCopiaAnt,
                      LPSTR lpLinhaCmd, int nCmdExibir)
```

Como você pode ver, a função *WinMain* retorna um valor *int*, exatamente como muitos de seus programas C++ fizeram. Isto, no entanto, é onde a similaridade termina (porém, como você aprenderá, a função de cabeçalho *WinMain* efetua processamento similar àquele do cabeçalho *main*). A palavra-chave *APIENTRY* indica que o usuário somente pode disparar (ou executar) o programa a partir de dentro do Windows. A Tabela 1265 detalha os parâmetros que seus programas precisam fornecer com a função *WinMain*.

Tabela 1265 Os parâmetros que *WinMain* aceita.

Tipo de Parâmetro	Nome do Parâmetro	Descrição
HINSTANCE	*hCopia*	O indicativo da ocorrência do aplicativo. Cada ocorrência tem um indicativo de ocorrência exclusivo. Seus programas usarão valores *hCopia* como um argumento para várias funções do Windows, e também podem usar o valor *hCopia* para distinguir entre múltiplas ocorrências de um determinado aplicativo.
HINSTANCE	*hCopiaAnt*	O indicativo anterior da ocorrência de um aplicativo. Esse valor será *NULL* se essa for a primeira ocorrência. Para o Windows 95, esse valor é sempre NULL.
LPSTR	*lpLinhaCmd*	Um ponteiro *far* para uma linha de comando terminada por *NULL*. Especifique o valor *lpLinhaCmd* ao chamar o aplicativo a partir do gerenciador de programas ou a partir de uma chamada a *WinExec*. Observe que, sob o Windows 95, isso é um ponteiro para a linha de comando inteira, não uma matriz de ponteiros para cada argumento (para que seus programas possam analisar a linha de comando antes de começar a processá-las).
int	*nExibeCmd*	Um inteiro que especifica a exibição da janela do aplicativo. Passe esse valor para *ShowWindow*.

No programa *generico.cpp*, a primeira ação de *WinMain* é definir uma variável do tipo *MSG*, uma variável do tipo *HWND* e uma variável do tipo *WNDCLASS*, como mostrado aqui:

```
MSG msg;
HWND hJan;
WNDCLASS cj;
```

MSG é um tipo enumerado sobre o qual você aprenderá mais em dicas posteriores. *HWND* é um indicativo de janela. O tipo *WNDCLASS* mantém informações sobre a classe da janela que o programa usa, como detalham as Dicas 1267 e 1269. Os próximos comandos dentro do programa *generico.cpp* são todos atribuições a valores-membro da variável *cj*. A Dica 1269 explicará as atribuições em detalhes. As dicas a seguir explicarão os comandos restantes dentro da função *WinMain*.

COMPREENDENDO A CRIAÇÃO DE JANELA 1266

Como você aprendeu, o fundamento da interação de qualquer programa Windows com o usuário é por meio de uma janela que o programa cria. Todo programa que você projeta no Windows para interagir com o usuário por meio da interface do Windows criará pelo menos uma janela durante seu processamento. A maioria dos programas registrará um conjunto de *classes de janela*, e criará múltiplas ocorrências dessas classes. Você aprenderá mais sobre o registro de classes de janela na Dica 1269.

Criar uma janela é um processo relativamente simples: você precisa primeiro determinar os componentes e a aparência da janela e depois usar a função *CreateWindow* da API Win32 para criar a janela. Como você viu na Dica 1257, seus programas implementarão a função *CreateWindow* com uma série de parâmetros. A forma generalizada da função *CreateWindow* é como segue:

```
HWND CreateWindow (LPCTSTR lpszNomeClasse,
                   LPCTSTR lpszNomeJanela, DWORD dwEstilo,
                   int x, int y, int nLarg, int nAltura,
                   HWND hjanMae, HMENU hmenu, HANDLE hcop,
                   LPVOID lpvParam)
```

Claramente, a função *CreateWindow* requer alguma preparação dentro de seu programa antes que o programa possa chamar a função. A Tabela 1266 detalha os onze parâmetros que você precisa passar para a função *CreateWindow*.

Tabela 1266 Os parâmetros para a função *CreateWindow*.

Tipo de Parâmetro	Nome do Parâmetro	Descrição
LPCTSTR	*lpszNomeClasse*	Um ponteiro constante para uma string terminada por *NULL* que contém um nome de classe de janela válido. O nome da classe pode ser um que o programa cria com *RegisterClass* ou um tipo de janela predefinido, como detalha a Dica 1269
LPCTSTR	*lpszNomeJanela*	Um ponteiro constante para uma string terminada por *NULL* que contém o nome da janela. Dependendo do estilo da janela, o nome da janela pode aparecer em várias posições diferentes.
DWORD	*dwEstilo*	Um valor WORD (um inteiro sem sinal de 32 bits) que corresponde aos estilos possíveis para uma janela. A Dica 1275 discute o tipo DWORD em detalhes. Você criará estilos a partir de valores que seus programas combinam junto com um operador *OR* binário. Por exemplo, um estilo aceitável seria *WS_CHILD	ES_LEFT*. As Dicas 1279 e 1280 discutem os estilos de janela em detalhes.
int	*x*	A posição horizontal do canto superior esquerdo da janela. Se a posição não for importante, seu programa deverá passar o valor *CW_USEDEFAULT* para o parâmetro *x*.
int	*y*	A posição vertical do canto superior esquerdo da janela. Se a posição não for importante, seu programa deverá passar o valor *CW_USEDEFAULT* para o parâmetro *y*.
int	*nLarg*	A largura horizontal da janela. Se a largura não for importante, seu programa deverá passar o valor *CW_USEDEFAULT* para o parâmetro *nLarg*.
int	*nAlt*	A altura vertical da janela. Se a altura não for importante, seu programa deverá passar o valor *CW_USEDEFAULT* para o parâmetro *nAlt*.
HWND	*hjanMae*	Um indicativo para a janela-mãe da janela. Se não houver uma janela-mãe, passe um valor *NULL* para esse parâmetro.
HMENU	*hmenu*	Um indicativo para o menu de uma janela. Passe *NULL* se a API usar o menu registrado com a classe da janela.
HANDLE	*hcop*	O indicativo da cópia (ocorrência) do programa que cria o controle.
LPVOID	*hpvParam*	Um ponteiro para dados que a função *CreateWindow* deve passar na mensagem *WM_CREATE*. Para janelas-filha da Interface de Múltiplos Documentos (MDI), o valor *lpvParam* deve ser um ponteiro para uma estrutura *CLIENTCREATESTRUCT*. Para a maioria das janelas cliente não-MDI, passe um valor *NULL*.

Embora a criação de uma janela possa parecer tenebrosa neste momento, você descobrirá, na dica a seguir, que muitas janelas que seus programas criam compartilharão características comuns — de modo que usar *CreateWindow* se tornará significativamente mais fácil.

1267 COMPREENDENDO A FUNÇÃO CREATEWINDOW

A dica anterior apresentou os fundamentos subjacentes à criação de janelas dentro de seus programas. No entanto, com seus onze parâmetros, compreender a função *CreateWindow* é significativamente mais fácil quando

você considera um caso real, em vez do caso geral descrito na dica anterior. Portanto, considere o fragmento de código a seguir do programa *generico.cpp*:

```
hJan = CreateWindow(lpszNomeApl, lpszTitulo,
                    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
                    CW_USEDEFAULT, 0, NULL, NULL, hCopia,
                    NULL);
if(!hJan)
    return false;
```

Os três comandos executam dois passos. O primeiro comando tenta criar uma janela. Se o comando for bem-sucedido, ele retornará um indicativo para a janela recém-criada. Se não for bem-sucedida, *CreateWindow* retornará *False*. O segundo comando verifica o indicativo da janela para determinar se a criação da janela foi bem-sucedida. Se a criação não foi bem sucedida, o terceiro comando termina o programa com um resultado falso (falha).

No entanto, compreender a criação de uma janela é mais complexo. A variável *lpszNomeApl* aponta para uma string que contém o nome do aplicativo — um nome de aplicativo que também corresponde a certas informações dentro do arquivo de recursos, como você aprendeu em dicas anteriores. O ponteiro *lpszTitulo* corresponde à string "Aplicativo Genérico", que a janela exibirá em sua barra de título. O parâmetro *WS_OVERLAPPEDWINDOW* diz à função *CreateWindow* para criar uma janela sobreposta — um estilo sobre o qual você aprenderá mais na Dica 1279. Os próximos quatro comandos informam a função *CreateWindow* de onde e de que tamanho criar a janela. O primeiro parâmetro *NULL* permite que *CreateWindow* saiba que essa janela não tem mãe; e o segundo parâmetro *NULL* diz a *CreateWindow* que deve usar o menu padrão para essa classe de janela. O penúltimo parâmetro passa na ocorrência do programa, e o parâmetro final passa um valor *NULL* para a mensagem *WM_CREATE*.

Especialmente nos seus primeiros programas, muitos desses parâmetros serão padrão ou usarão variáveis que seu programa inicializa em sua inicialização, para manter o processamento mais simples. Na verdade, o único parâmetro que você provavelmente modificará com regularidade durante seus primeiros esforços em programação para o Windows no parâmetro *dwEstilo* (estilo da janela).

COMPREENDENDO A FUNÇÃO SHOWWINDOW 1268

Na dica anterior você aprendeu como o programa *generico.cpp* cria uma janela. No entanto, quando você programa dentro do Windows, precisa tornar a janela visível após criá-la. Para ajudar seus programas a exibir janelas, a API Win32 fornece a função *ShowWindow*. Seus programas usarão a função *ShowWindow* como mostrado no seguinte protótipo:

```
BOOL ShowWindow(HWND hJan, int nCmdExibir);
```

A função *ShowWindow* retorna um valor de sucesso True ou False que seus programas devem verificar. O parâmetro *hJan* é o indicativo da janela que você quer exibir. No programa *generico.cpp*, *hJan* refere-se à janela recém-criada. O parâmetro *nCmdExibe* controla como seu programa exibe a janela. O valor que você passa no parâmetro *nCmdExibir* precisa corresponder a um dos valores listados na Tabela 1268.

Tabela 1268 Valores válidos para o parâmetro ***nCmdExibe***.

Valor	Significado
SW_HIDE	Oculta a janela.
SW_MINIMIZE	Minimiza a janela e ativa a janela de alto nível na lista de janela do sistema.
SW_RESTORE	Ativa e exibe a janela. Se a janela estiver atualmente minimizada ou maximizada, *ShowWindow* retornará a janela às suas dimensões e sua posição originais.
SW_SHOW	Exibe a janela em seu tamanho e posição atuais
SW_SHOWDEFAULT	Exibe a janela em seu estado de aplicativo padrão. *ShowWindows* obtém o estado padrão do aplicativo a partir da estrutura *STARTUPINFO*, sobre a qual você aprenderá em dicas posteriores.

Tabela 1268 Valores válidos para o parâmetro ***nCmdExibe***. (*Continuação*)

Valor	Significado
SW_SHOWMAXIMIZED	Exibe a janela como maximizada.
SW_SHOWMINIMIZED	Exibe a janela como minimizada como um ícone.
SW_SHOWMINNOACTIVE	Exibe a nova janela minimizada, e a janela atualmente ativa permanecerá ativa.
SW_SHOWNA	Exibe a janela em seu estado atual, e a janela ativa permanecerá ativa.
SW_SHOWNOACTIVE	Exibe a janela em seu tamanho e posição mais recentes, e a janela ativa permanecerá ativa.
SW SHOWNORMAL	Exibe a janela em seu tamanho normal.

A modificação a seguir no programa *generico.cpp*, *tst_max.cpp*, instrui o Windows a maximizar a janela de aplicativo quando o usuário selecionar a opção Testar no menu Arquivo. Exclua o código atual dentro da função *ProcJan* e o substitua com o código a seguir:

```
LRESULT CALLBACK ProcJan ( HWND hJan, UINT uMsg, WPARAM wParam, LPARAM lPa-
ram )
 {
   switch(uMsg)
   {
      case WM_COMMAND :
        switch(LOWORD(wParam))
        {
           case IDM_TESTAR :
              ShowWindow(hJan, SW_SHOWMAXIMIZED);
              break;
           case IDM_SAIR :
              DestroyWindow(hJan);
              break;
        };
        break;
     case WM_DESTROY :
        PostQuitMessage(0);
        break;
    'default :
        return(DefWindowProc(hJan, uMsg, wParam, lParam));
   }
   return(0L);
 }
```

O único resultado a partir da mudança do código do programa *generico.cpp* original é que a janela se maximiza para preencher a tela quando o usuário selecionar *Testar!* no menu da janela. Após você compilar e executar o programa *generico.cpp* modificado, teste o processamento do programa maximizando primeiro a janela, e, depois, definindo-a de volta ao seu tamanho normal. Em dicas posteriores, você fará manipulações mais extensas para controlar a aparência da janela e usará as definições para *ShowWindow* para controlar melhor a aparência da janela.

1269 COMPREENDENDO A FUNÇÃO REGISTERCLASS

Como você aprendeu nas Dicas 1266 e 1267, quando seus programas criam janelas, eles podem criar uma janela de uma classe predefinida ou podem criar seus próprios estilos de janela. Quando você criar seus próprios estilos de janela, é preciso registrar o estilo da janela com o Windows antes de poder usar o estilo para criar janelas. Você usará a função da API *RegisterClass* para registrar os estilos da janela. Você usará a função *RegisterClass* dentro de seus programas, como mostrado no protótipo a seguir:

```
ATOM RegisterClass(CONST WNDCLASS* lpcw)
```

Como você pode ver, a função *RegisterClass* retorna um valor do tipo *ATOM*. Um *ATOM* é um valor *WORD* que se refere a strings de caracteres de uma maneira que não distingue a caixa das letras. O fato de *ATOMs* se referirem a strings de um modo a não distinguir a caixa significa que "jamsa" é igual a "JAMSA" — uma igualdade que, como você sabe, normalmente não é verdadeira em C++. O Windows armazena *ATOMs* em uma tabela de *ATOMs* — de modo que o valor *WORD* que um *ATOM* mantém é na verdade muito similar a um indicativo.

Além de retornar um valor *ATOM*, a função *RegisterClass* aceita um único parâmetro — um ponteiro constante a uma estrutura do tipo *WNDCLASS*. O Windows define a estrutura *WNDCLASS*, como mostrado aqui:

```
typedef struct tagWNDCLASS
 {
   UINT        style;
   WNDPROC     lpfnWndProc;
   int         cbClsExtra;
   int         cbWndExtra;
   HINSTANCE   hInstance;
   HICON       hIcon;
   HCURSOR     hCursor;
   HBRUSH      hbrBackground;
   LPCTSTR     lpszMenuName;
   LPCSTR      lpszClassName;
 } WNDCLASS;
```

Como você pode ver, dez membros de dados compõem a estrutura *WNDCLASS*. A Tabela 1269.1 detalha os membros de dados da estrutura *WNDCLASS*.

Tabela 1269.1 Os membros de dados da estrutura ***WNDCLASS***.

Nome do Membro	Tipo	Função
style	*UINT*	O parâmetro *style* precisa ser um ou mais dos estilos detalhados na Tabela 1269.2, combinados com o operador binário OU.
lpfnWndProc	*WNDPROC*	Aponta para uma função de *callback* de janela que processa as mensagens que o Windows gera para a janela.
cbClsExtra	*int*	O número de bytes extra que *RegisterClass* deverá alocar no final da estrutura de classe da janela para armazenar informações.
cbWndExtra	*int*	O número de bytes extra que *RegisterClass* deve alocar após a criação de cada ocorrência para armazenar informações
hInstance	*HINSTANCE*	Um indicativo para a ocorrência da qual a classe da janela faz parte.
hIcon	*HICON*	Um indicativo para o ícone que *CreateWindow* usará para essa classe de janela.
hCursor	*HCURSOR*	Um indicativo para o cursor que *CreateWindow* usará para essa classe de janela.
hBrush	*HBRUSH*	Um indicativo para um pincel que *CreateWindow* usará para criar o fundo da janela. Você aprenderá mais sobre os pincéis em dicas posteriores.
lpszMenuName	*LPCTSTR*	Um ponteiro para uma constante string terminada por *NULL* com o nome de menu padrão para a classe. Você deverá definir esse valor como *NULL* se a janela não tiver nenhuma classe de menu padrão.
lpszClassName	*LPCTSTR*	Um ponteiro para uma constante string terminada por *NULL* que contém o nome da classe. Seus programas mais tarde usarão o nome da classe no parâmetro *lpszClassName* da função *CreateWindow*.

Como indica a Tabela 1269.1, o membro de dados *style* da classe da janela pode corresponder a uma ou mais constantes binárias. A Tabela 1269.2 lista os valores aceitáveis para o membro de dados *style*.

Tabela 1269.2 Valores válidos para os membros de dados *style* da classe da janela.

Constante de Estilo	Significado
CS_BYTEALIGNCLIENT	Alinha a área cliente de uma janela no limite de byte horizontalmente para aumentar o desempenho ao desenhar. Isso afeta a largura da janela e sua posição horizontal na tela.
CS_BYTEALIGNWINDOW	Alinha uma janela em um limite de byte horizontalmente.
CS_CLASSDC	Aloca um dispositivo de contexto (DC) que todas as janelas na classe compartilharão. Se múltiplos encadeamentos tentarem acessar o dispositivo do contexto simultaneamente, o Windows apenas permitirá que um encadeamento termine com sucesso. Você aprenderá mais sobre os dispositivos do contexto em dicas posteriores.
CS_DBLCLKS	Notificará uma janela quando o usuário der um clique duplo no botão do mouse.
CS_GLOBALCLASS	Cria uma classe que está disponível a todos os aplicativos enquanto o aplicativo que criou a classe estiver aberto. Geralmente, você usará essa definição quando criar controles personalizados para outros programas usarem.
CS_HREDRAW	Redesenha a janela inteira se o usuário ajusta o tamanho horizontal.
CS_NOCLASE	Desabilita o comando Fechar no menu Sistema.
CS_OWNDC	Aloca um dispositivo do contexto único para cada ocorrência da classe da janela.
CS_PARENTDC	Cada janela que seus programas criarem da classe usará o dispositivo do contexto da janelá-mãe.
CS_SAVEBITS	Salva, como um mapa de bits, a porção da imagem da tela que uma janela obscurece. O Windows usa o mapa de bits para recriar a imagem na tela quando o usuário remover a janela.
CS_VREDRAW	Redesenha a janela inteira se o usuário ajustar o tamanho vertical.

Agora que você compreende como o comando *RegisterClass* funciona, considere as seguintes atribuições do programa *generico.cpp*, que inicializam a classe da janela que o programa utiliza:

```
cj.style          = CS_HREDRAW | CS_VREDRAW;
cj.lpfnWndProc    = (WNDPROC)ProcJan;
cj.cbClsExtra     = 0;
cj.cbWndExtra     = 0;
cj.hInstance      = hInst;
cj.hIcon          = LoadIcon(hCopia, lpszNomeAplic);
cj.hCursor        = LoadCursor(NULL, IDC_ARROW);
cj.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
cj.lpszMenuName   = lpszNomeAplic;
cj.lpszClassName  = lpszNomeAplic;
```

A primeira atribuição informa ao sistema operacional para redesenhar a janela inteira sempre que o usuário redefinir o tamanho da janela em ambas as direções. A segunda atribuição diz ao sistema operacional que *ProcJan* é a função de *callback*. Os dois próximos comandos de atribuição dizem a *RegisterClass* para não alocar espaço extra, e o comando *cj.hInstance*= instrui o compilador a usar a ocorrência do programa atual.

Os dois próximos comandos de atribuição (para *hIcon* e *hCursor*) carregam o ícone e o cursor para a janela usar. O comando de atribuição após esses comandos (*hbrBackground*=) cria um indicativo para um pincel colorido, e os dois comandos finais atribuem o nome de menu padrão da janela e o nome de classe da janela.

Novamente, como verá com muitos dos comandos e estruturas que você manipulará para o Windows, com freqüência você usará os comandos e estruturas do mesmo modo, mais frequentemente com os mesmos valores e apenas de forma ocasional modificam alguns valores quando você criar uma janela especial.

Nota: Um caso especial existe quando você usa CreateWindow para criar uma janela usando uma classe existente. Os nomes das classes existentes são BUTTON, LISTBOX, COMBOBOX, STATIC, EDIT, MDICLIENT e SCROLLBAR. Não é necessário registrar essas classes antes de seu programa criar uma janela usando uma das classes.

APRENDENDO MAIS SOBRE AS MENSAGENS 1270

A última parte da função *WinMain* no programa *generico.cpp* é o laço *while*, que processa mensagens do sistema. Como aprendeu, como regra, você escreverá seus programas Windows para ficar continuamente em execução, até que o usuário de forma específica instrua a janela a fechar. Cada programa Windows que você escrever usará um *laço de mensagem* para continuar as mensagens de processamento até que o usuário instrua o programa a parar. A forma padrão do laço de mensagem é mostrado aqui:

```
while(GetMessage(&msg, NULL, 0, 0))
 {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
 }
```

Nas Dicas 1271 e 1272, você aprenderá mais sobre as funções *TranslateMessage* e *DispatchMessage*. No entanto, é importante que primeiro compreenda *GetMessage* e o que ela retorna. Você usará a função *GetMessage* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL GetMessage(LPMSG lpmsg, HWND hjan, UINT uMsgFiltroMin,
                UINT uMsgFiltroMax)
```

A função *GetMessage* retorna um valor verdadeiro ou falso. *GetMessage* retorna verdadeiro até que recupere a mensagem *WM_QUIT*. A Tabela 1270 lista os parâmetros para a função *GetMessage*.

Tabela 1270 Os parâmetros para a função ***GetMessage***.

Nome da Função	Tipo	Propósito
lpmsg	*MSG*	Retorna um ponteiro para uma estrutura *MSG* Esta dica define a estrutura *MSG* após essa tabela.
hjan	*HWND*	Um indicativo para a janela que recebe as mensagens. Normalmente, você definirá esse valor como *NULL*, o que instrui *GetMessage* a pegar todas as mensagens para o encadeamento atual.
uMsgFiltroMin	*UINT*	O valor de mensagem mínimo a receber. Normalmente, você definirá esse parâmetro como 0.
uMsgFiltroMax	*UINT*	O valor de mensagem máximo a receber. Se você definir *uMsgFiltroMin* e *uMsgFiltroMax* como 0, *GetMessage* recuperará todas as mensagens.

Conforme indica a Tabela 1270, a função *GetMessage* recebe um parâmetro do tipo *MSG*. O Windows define o tipo *MSG* no arquivo *winuser.h*, como mostra aqui:

```
typedef struct tagMSG {
 HWND        hwnd;                //window handle
 UINT        message;             //message ID
 WPARAM      wParam;              //wParam value
 LPARAM      lParam;              //lParam value
 DWORD       time;                //milliseconds since startup
 POINT       pt;                  //screen coordinates of current mouse location
 } MSG;
```

Embora cada componente da estrutura *MSG* seja importante, o qual mais freqüentemente você manipulará será o membro *message*, que corresponde a uma das muitas definições de constante do Windows para as mensagens. Você aprenderá mais sobre as constantes de mensagem em dicas posteriores.

1271 USANDO TRANSLATEMESSAGE PARA PROCESSAR AS MENSAGENS

Como você aprendeu na dica anterior, as funções *WinMain* de seu programa geralmente terminarão com um laço *while*, que recupera mensagens até que o usuário envie a mensagem *WM_QUIT* para o sistema. Dentro do laço de mensagem, como você viu na dica anterior, o programa primeiro chama a função *TranslateMessage*. Depois, *TranslateMessage* pega uma mensagem de tecla virtual (tal como *VK_TAB*) que o sistema gera quando o usuário pressiona uma tecla e envia o código *WM_CHAR* correspondente para a fila de mensagem do aplicativo (*WM_CHAR* significa *Windows Message Character*, ou Mensagem de Caractere do Windows). Se a mensagem não for uma mensagem de tecla virtual, *WM_CHAR* retornará falso e não processará a mensagem. Você usará a função *TranslateMessage* dentro de seus programas como mostrado no protótipo a seguir:

```
BOOL TranslateMessage(CONST MSG*, lpMsg);
```

Como visto, geralmente você chamará *TranslateMessage* imediatamente após uma chamada a *GetMessage*, embora você também possa usar uma mensagem que a função *PeekMessage* (explicada em dicas posteriores) retorna, como mostrado aqui:

```
while(GetMessage(&msg, NULL, 0, 0))
  {
     TranslateMessage(&msg);
     DispatchMessage(&msg);
  }
```

1272 USANDO DISPATCHMESSAGE PARA PROCESSAR AS MENSAGENS

Normalmente, a função *WinMain* dentro de seus programas fica em um laço, aguardando as mensagens. Quando uma mensagem chega, *WinMain* despacha a mensagem para outra função que processa a mensagem. Como você aprendeu na Dica 1270, seus programas criarão um laço de mensagem para processar mensagens até que o programa termine. O último componente do laço de mensagem é a função *DispatchMessage*, que envia a mensagem para a função de processamento após o comando *TranslateMessage* garantir que a mensagem é uma mensagem do Windows. A função de processamento que *DispatchMessage* chama é a função de *callback* que a classe de janela define em seu registro. Você implementará a função *DispatchMessage*, como mostrado na seguinte forma generalizada:

```
long DispatchMessage(CONST MSG* lpmsg);
```

Embora *DispatchMessage* retorne um valor do tipo *long*, seus programas geralmente ignorarão o resultado da chamada *DispatchMessage* porque ele não fornece informações úteis para o seu programa.

Nota: Seus laços de mensagem deverão incluir a função DispatchMessage ou não poderão processar as mensagens que o sistema envia.

1273 COMPREENDENDO OS COMPONENTES DE UM PROGRAMA WINDOWS SIMPLES

Nas 16 dicas anteriores, você estudou o programa *generico.cpp* e seus componentes. O programa *generico.cpp* inclui todos os componentes básicos de qualquer programa Windows. Quando você projetar programas Windows, precisará garantir que seus programas Windows incluirão todos os seguintes componentes, exatamente como faz o programa *generico.cpp*.

- Um *Arquivo de Recursos:* Embora você possa escrever programas Windows sem usar um arquivo de recursos, isso lhe criará trabalho adicional, e não é a construção de programa Windows padrão. Todo programa Windows que você escrever deverá incluir um arquivo de recurso. No entanto, você não deverá criar mais de um arquivo de recurso por programa. Lembre-se, um arquivo de recurso pode incluir informações sobre muitas janelas e seus conteúdos dentro de um determinado programa, de modo que você deve colocar todas as informações para cada corpo de programa dentro de um único arquivo.
- O cabeçalho *windows.h:* Todos os programas Windows precisam incluir o cabeçalho *windows.h* que, por sua vez, inclui todos os arquivos de cabeçalho necessários para tipos, funções e classes específicos do Windows.
- A função *WinMain:* Exatamente como todos os seus programas Windows requeriam a função *main*, todos os seus programas para o Windows precisam incluir a função *WinMain*. No entanto, lembre-se de que você não pode ter uma função *WinMain* sem seus quatro parâmetros esperados: *HINSTANCE hCopia, HINSTANCE hCopAnterior, LPSTR lpLinhaCmd* e *int nCmdExibir*.
- Um laço de mensagem do Windows: Todo programa Windows que requer interação do usuário (em resumo, quase todos os programas Windows) processa mensagens dentro de um laço de mensagem. O laço de mensagem recupera mensagens a partir da fila de mensagem do sistema e os processa dentro de seus programas.
- Uma função de *callback* de mensagem: Quando você criar uma janela, um dos parâmetros para essa janela é onde ela deverá enviar suas mensagens dentro do programa — a função de *callback* de mensagem O laço de mensagem também envia mensagens para a função de *callback*. *Todo programa Windows precisa ter uma função de callback de mensagem.* Você normalmente nomeará suas funções de *callback* de forma consistente. Você também pode ter múltiplas funções de *callback* de mensagem para tratar as mensagens dentro de diferentes janelas de programa de forma diferente.

Praticamente todo programa Windows terá esses cinco componentes. Todo programa Windows que você usar dentro deste livro terá todos os cinco componentes. Se você projetar um programa sem todos os cinco componentes, provavelmente terá dificuldade em fazer o programa trabalhar de forma correta sob o Windows.

COMPREENDENDO O TIPO LPCTSTR 1274

Como você aprendeu, seus programas usarão o tipo *LPCTSTR* do Windows para armazenar um ponteiro de 32 bits para uma string de caracteres constante. O tipo caractere *LPCTSTR* é portátil tanto para o Unicode quanto para o conjunto de caractere de byte duplo (DBCS). Você com mais freqüência declarará as strings como *LPCTSTR* se a string for um parâmetro para uma função que a função não modificará. Por questões de velocidade e conveniência, se você souber que uma função recebe parâmetros de string *por valor*, deverá declarar o parâmetro da função string como tipo *LPCTSTR*. Seus programas deverão usar um parâmetro *LPTSTR* (um ponteiro de 32 bits para uma string terminada por *NULL*) em vez de um parâmetro *LPCTSTR* se a função puder retornar valores dentro do parâmetro.

Por exemplo, o fragmento de código a seguir declara uma classe *CName*. A classe *CName* inclui as duas funções-membro, *SetData* e *GetData*. Como a função *SetData* não altera as informações dentro de suas strings componentes, *SetData* declara ambas as strings como *LPCTSTR*. *GetData*, por outro lado, modifica as string e assim declara as strings como *LPSTR*, como mostrado aqui:

```
class CName
 {
private :
   LPSTR m_primNome;
   char m_meioInit;
   CString m_sobrenome;
public :
   CName()  {}
   void SetData( LPCTSTR fn, const char mi, LPCTSTR in )
   {
      m_primNome = fn;
```

```
        m_meioInit = mi;
        m_sobrenome = ln;
    }
    void GetData( LPSTR& cfn, char mi, LPSTR& cln )
    {
        cfn = m_primNome;
        mi = m_meioInit;
        cln = m_sobrenome;
    }
};
```

1275 COMPREENDENDO O TIPO DWORD

Como você aprendeu em dicas anteriores, uma *DWORD* é um tipo *WORD* duplo. Um tipo *WORD* é um inteiro de 16 bits sem sinal (tornando-o equivalente ao *unsigned long int* em C de 16 bits). Portanto, uma *WORD* dupla é, então, um inteiro de 32 bits sem sinal capaz de armazenar valores até 2^{32}-1. A Figura 1275.1 mostra como seu computador aloca memória para uma *WORD* e uma *DWORD*.

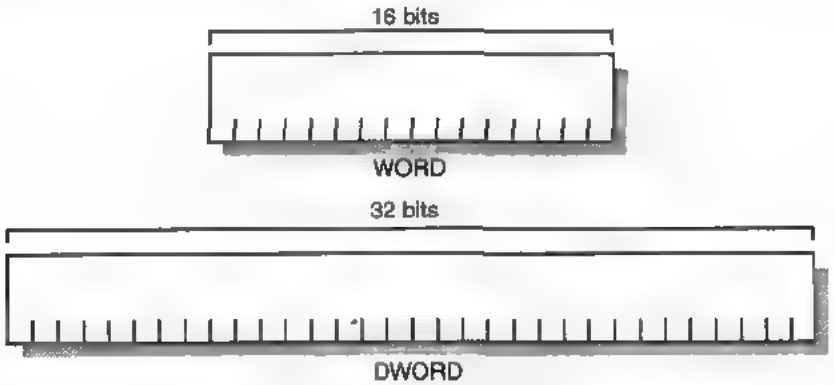

Figura 1275.1 Alocação de memória para uma *WORD* e uma *DWORD*.

Embora seus programas possam usar um valor *DWORD* para armazenar números inteiros grandes não-sinalizados, seus programas mais freqüentemente usam valores *DWORD* para manter um endereço de segmento e de deslocamento de 32 bits. Como duas *WORDs* compõem uma *DWORD*, a maioria dos compiladores C++ fornece várias ferramentas para quebrar *DWORDs* em uma *WORD* mais significativa e uma *WORD* menos significativa (que contém os dois bytes menos significativos). Quando você quebra uma *DWORD* em duas *WORDs*, a *WORD* mais significativa representa o endereço de segmento e a *WORD* menos significativa representa o endereço de deslocamento, como mostrado na Figura 1275.2.

Figura 1275.2 A *DWORD* pode representar um endereço de segmento e de deslocamento.

1276 COMPREENDENDO AS CLASSES PREDEFINIDAS DO WINDOWS

Como as Dicas 1267 e 1269 mencionaram rapidamente, você também pode criar janelas de vários tipos derivados dentro de seus programas usando as classes predefinidas do Windows. As janelas que você criará usando as classes de janela predefinida do Windows, com exceção da janela *MDICLIENT* (sobre a qual você aprenderá em dicas posteriores), são conhecidas como controles. Apesar de você não aprender outros métodos neste livro,

também poderá usar os controles ActiveX e classes de janela específicas do compilador para criar janelas adicionais dentro de seus programas. A API do Windows suporta as classes de janela predefinidas listadas na Tabela 1276.

Tabela 1276 As classes de janela preexistentes.

Classe	Descrição
BUTTON	Seus programas usarão o controle *BUTTON* para criar botões dentro de suas janelas. Os botões podem ser botões retangulares, caixas de grupo, caixas de seleção, botões de opção ou janelas de ícone. Geralmente, seus programas usarão um botão para iniciar o processamento de um evento, ou o usuário usará um botão para informar o sistema de que completou a entrada de dados no formulário que a janela exibe. A Figura 1276.1 mostra um botão em uma janela.

Figura 1276.1 Um botão em uma janela.

LISTBOX	Seus programas usarão caixas de lista para manter listas de informações para os usuários. Um controle *LISTBOX* difere de um controle *COMBOBOX*, pois não aceita entrada além das seleções dentro da lista que o *LISTBOX* contém. A Figura 1276.2 mostra um controle *LISTBOX* de exemplo.

Figura 1276.2 Exemplo de um controle *LISTBOX*.

COMBOBOX	Um controle *COMBOBOX* é um controle *EDIT* combinado com um controle *LISTBOX*. Dentro de seus programas, você geralmente usará caixas de combinação para permitir que seus usuários selecionem a partir de uma lista e insiram suas próprias novas entrada na lista. A Figura 1276.3 mostra um controle *COMBOBOX* de exemplo.

Figura 1276.3 Exemplo de um controle *COMBOBOX*.

STATIC	Seus programas freqüentemente usarão um controle *STATIC*, também conhecido como rótulo, para colocar informações na área cliente da sua janela. As informações que você colocará dentro de um controle *STATIC* deverão ser informações que o usuário não editará ou modificará durante o curso da execução do programa. A Figura 1276.4 mostra um controle *STATIC* de exemplo.

Tabela 1276 As classes de janela preexistentes. (*Continuação*)

Classe	Descrição

Figura 1276.4 Exemplo de um controle *STATIC*.

EDIT	Você usará controles *EDIT* para permitir que os usuários insiram informações dentro de seus programas por meio da interface do Windows Geralmente, a maioria dos programas Windows incluirá um ou mais controles *EDIT*, especialmente programas de aplicativos comerciais. A Figura 1276.5 mostra um exemplo de um controle *EDIT*.

Figura 1276.5 Exemplo de um controle *EDIT*.

SCROLLBAR	Seus programas usarão o controle *SCROLLBAR* para colocar barras de rolagem nas bordas das janelas existentes, dentro da área cliente da janela, para receber entrada de dados do usuário.

Na Dica 1277, você criará um programa simples que usa várias das classes de janela predefinidas. Em geral, seus programas normalmente usarão muitas classes de janela predefinidas dentro de um único programa e, com freqüência usarão muitas classes diferentes dentro de uma única janela.

Nota: *O Microsoft Visual C++ e o Borland C++ 5.02, os dois compiladores mais usados no desenvolvimento em C++ para o Windows, incluem suporte para o controle de arrastar-e-soltar, o que lhe permite acrescentar controles nas suas janelas sem seguir os passos adicionais necessários para criar os controles usando o comando CreateWindow.*

1277 USANDO CLASSES PREDEFINIDAS PARA CRIAR UMA JANELA SIMPLES

Na dica anterior, você aprendeu sobre as classes predefinidas que o comando *CreateWindow* suporta. Como você aprendeu, seus programas geralmente usarão classes predefinidas para criar janelas-filha (controles) dentro de uma janela de programa. Usar classes predefinidas não é mais difícil do que criar uma ocorrência de uma janela que seus programas já registraram. Seus programas simplesmente chamarão a função *CreateWindow* com o primeiro parâmetro que contém o nome da classe predefinida, o segundo parâmetro que contém o texto dentro do controle predefinido, e os últimos parâmetros que determinam o estilo do controle e a colocação na janela do programa.

Por exemplo, o CD-ROM que acompanha este livro inclui o programa *exibe_3.cpp*, que cria a janela de abertura genérica que você usou anteriormente no início do programa. Quando o usuário selecionar a opção *Testar!*, o programa exibirá um controle *STATIC*, um controle *EDIT* e um controle *BUTTON*. O programa *exibe_3* usa três arquivos: *exibe_3.h*, *exibe_3.cpp* e *exibe_3.rc*. Em geral, os arquivo são os mesmos que para o programa *generico.cpp*, com exceção da função *ProcJan*, que chama *CreateWindow* três vezes quando o usuário seleciona a opção *Testar!*.

Como você verá a partir da listagem do código, a única modificação entre a função *ProcJan* no arquivo *exibe_3.cpp* e a função *ProcJan* no programa *generico.cpp* é como o programa *exibe_3.cpp* trata a seleção do usuário da opção *Testar!*. Dentro do código, se o usuário selecionar *Testar!*, que envia uma mensagem *IDM_TESTAR*

para *ProcJan*, o código chamará *CreateWindow* três vezes: uma com um controle *STATIC*, uma com um controle *EDIT* e uma com um controle *BUTTON*.

O WINDOWS ENVIA WM_CREATE QUANDO CRIA UMA JANELA 1278

Como você aprendeu, sempre que o Windows efetua uma atividade significativa, ele envia uma mensagem para os programas que estão em execução atualmente, informando esses programas da atividade que o Windows completou. Criar janelas não é exceção. Cada vez que um programa chama a função *CreateWindow*, o Windows envia cinco mensagens para a função *ProcJan*. Tipicamente, seus programas processarão a mensagem *WM_CREATE* dentro da função *ProcJan* e permitirão que as outras quatro mensagens passem para *DefWindowProc*. Seus aplicativos usarão a mensagem *WM_CREATE* para informar o programa para inicializar a janela. A Tabela 1278 lista as cinco mensagens que o Windows envia para o programa quando cria uma janela com sucesso.

Tabela 1278 As mensagens que o Windows gera após seu programa chamar a função ***CreateWindows***.

Mensagem	Significado
WM_GETMINMAXINFO	Pega o tamanho e a posição da janela que o Windows está criando.
WM_NCCREATE	Indica que o Windows está prestes a criar a área não-cliente da janela. A função *DefWindowProc* aloca memória para a janela e inicializa as barras de rolagem quando recebe a mensagem *WM_NCCREATE*.
WM_NCCALCSIZE	A função *DefWindowProc* calcula o tamanho e a posição da área cliente da janela quando recebe a mensagem *WM_NCCALCSIZE*.
WM_CREATE	A função *WndProc* deve efetuar inicialização da janela quando ele recebe a mensagem *WM_CREATE*, que indica que o Windows está prestes a criar a janela.
WM_SHOWWINDOW	Informa a função *DefWindowProc* que o Windows está prestes a exibir a janela.

Como regra, seus programas devem permitir que a função *DefWindowProc* trate a maior parte do processamento quando você cria as janelas. (Na verdade, seus programas devem permitir que *DefWindowProc* trate todas as mensagens que você não requer especificamente que o programa trate.) Como viu, você deve tornar a chamada a *DefWindowProc* o padrão para o comando *case* dentro de sua função *ProcJan*, como mostrado aqui:

```
default :
   return(DefWindowProc(hJan, uMsg, wParam, lParam));
```

COMPREENDENDO OS ESTILOS WINDOW E CONTROL 1279

Como você aprendeu em dicas anteriores, seus programas chamarão a função *CreateWindow* com um conjunto de parâmetros que fornecem à função informações sobre a nova janela a criar. Um parâmetro que você passará em toda chamada da função *CreateWindow* é o parâmetro *DWORD dwStyle*. Como você aprendeu, seus programas criam o valor para o parâmetro *dwStyle* usando uma seqüência de operadores *OU* bit a bit com uma seqüência de constantes de estilo de janela. Como você viu rapidamente na Dica 1277, seus programas também usarão diferentes parâmetros *dwStyle* ao criarem ocorrências das classes de janela predefinidas. A Tabela 1279.1 lista os parâmetros *dwStyle* para as classes de janela.

Tabela 1279.1 Os valores possíveis para o parâmetro ***dwStyle*** quando você criar janelas dentro de seus programas.

Estilo	Descrição
WS_BORDER	Cria uma janela que tem uma borda de linha fina.
WS_CAPTION	Cria uma janela que tem uma barra de título (inclui o estilo *WS_BORDER*).
WS_CHILD	Cria uma janela-filha. Você não pode usar esse estilo com o estilo *WS_POPUP*

Tabela 1279.1 Os valores possíveis para o parâmetro ***dwstyle*** quando você criar janelas dentro de seus programas. (*Continuação*)

Estilo	Descrição
WS_CHILDWINDOW	Cria a mesma janela que o estilo *WS_CHILD*.
WS_CLIPCHILDREN	Exclui a área que as janelas-filha ocupam quando o desenho ocorre dentro da janela-mãe. Você poderá usar esse estilo ao criar a janela-mãe.
WS_CLIPSIBLINGS	Corta as janelas-filha relativas umas às outras; isto é, quando uma determinada janela-filha recebe uma mensagem *WM_PAINT*, o estilo *WS_CLIPSIBLINGS* corta todas as outras janelas-filha a partir da região da janela-filha que o programa estiver atualizando. Se seu programa não especificar *WS_CLIPSIBLINGS* e as janelas-filha se sobrescreverem, será possível, ao desenhar dentro da área cliente de uma janela-filha, acidentalmente desenhar dentro da área cliente de uma janela-filha vizinha.
WS_DISABLED	Cria uma janela que está inicialmente desabilitada. Uma janela desabilitada não pode receber entrada do usuário.
WS_DLGFRAME	Cria uma janela que tem uma borda com o estilo que você tipicamente usará com caixas de diálogo. Uma caixa com o estilo *WS_DLGFRAME* não tem uma barra de título.
WS_GROUP	Especifica o primeiro controle de um grupo de controles. O grupo consiste desse primeiro controle e de todos os controles definidos após ele, até o próximo controle com o estilo *WS_GROUP*. O primeiro controle em cada grupo normalmente tem o estilo *WS_TABSTOP* para que o usuário possa passar de grupo para grupo. O usuário pode subseqüentemente alterar o foco do teclado de um controle no grupo para o próximo controle no grupo usando as teclas de direção.
WS_HSCROLL	Cria uma janela que tem um barra de rolagem horizontal
WS_ICONIC	Cria uma janela inicialmente minimizada. O mesmo que o estilo *WS_MINIMIZE*.
WS_MAXIMIZE	Cria uma janela inicialmente maximizada.
WS_MAXIMIZEBOX	Cria uma janela que tem um botão *MAXIMIZE*. Seus programas não podem combinar o estilo *WS_MAXIMIZEBOX* com o estilo *WS_EX_CONTEXTHELP* Seus programas também precisarão especificar o estilo *WS_SYSMENU* quando eles especificarem o estilo *WS_MAXIMIZEBOX*.
WS_MINIMIZE	Cria uma janela inicialmente minimizada. O mesmo que o estilo *WS_ICONIC*.
WS_MINIMIZEBOX	Cria uma janela que tem um botão *MINIMIZE*. Seus programas não podem combinar o estilo *WS_MINIMIZEBOX* com o estilo *WS_ES_CONTEXTHELP* Seus programas também precisam especificar o estilo *WS_SYSMENU* com o estilo *WS_MINIMIZEBOX*.
WS_OVERLAPPED	Cria uma janela sobreposta Uma janela sobreposta tem uma barra de título e uma borda. O mesmo que o estilo *WS_TILED*.
WS_OVERLAPPEDWINDOW	Cria uma janela sobreposta que inclui os estilos *WS_OVERLAPPED*, *WS_CAPTION*, *WS_SYSMENU*, *WS_THICKFRAME*,. *WS_MINIMIZEBOX* e *WS_MAXIMIZEBOX*. O mesmo que o estilo *WS_TILEDWINDOW*.
WS_POPUP	Cria uma janela de ativação instantânea. Você não pode usar o estilo *WS_POPUP* com o estilo *WS_CHILD*.
WS_POPUPWINDOW	Cria uma janela de ativação instantânea que inclui os estilo *WS_BORDER*, *WS_POPUP* e *WS_SYSMENU*. Seus programas precisam combinar os estilos *WS_CAPTION* e *WS_POPUPWINDOW* para tornar visível o menu da janela.
WS_SIZEBOX	Cria uma janela que tem uma borda para ajustar o tamanho. O mesmo que o estilo *WS_THICKFRAME*.

Tabela 1279.1 Os valores possíveis para o parâmetro ***dwstyle*** quando você criar janelas dentro de seus programas. (*Continuação*)

Estilo	Descrição
WS_SYSMENU	Cria uma janela que tem um menu de sistema em sua barra de título. Seu programa também precisa especificar o estilo *WS_CAPTION*.
WS TABSTOP	Especifica o controle que pode receber o foco do teclado quando o usuário pressiona a tecla Tab Pressionar a tecla Tab muda o foco do teclado para o próximo controle com o estilo *WS_TABSTOP*.
WS_THICKFRAME	Cria uma janela que tem uma borda para ajustar o tamanho. O mesmo que o estilo *WS_SIZEBOX*.
WS_TILED	Cria uma janela sobreposta Uma janela sobreposta tem uma barra de título e uma borda. O mesmo que o estilo *WS_OVERLAPPED*.
WS_TILEDWINDOW	Cria uma janela sobreposta com os estilos *WS_OVERLAPPED,, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS MINIMIZEBOX* e *WS_MAXIMIZEBOX*. O mesmo que o estilo *WS_OVERLAPPEDWINDOW*.
WS_VISIBLE	Cria uma janela inicialmente visível.
WS_VSCROLL	Cria uma janela que tem uma barra de rolagem vertical.

Como você aprendeu, seus programas podem usar valores diferentes para as classes predefinidas do Windows quando você criar controles ou outros tipos de janelas. A Tabela 1279.2 detalha os valores possíveis para o parâmetro *dwStyle* quando você criar uma janela *BUTTON* dentro de seus programas.

Tabela 1279.2 Os valores possíveis para o parâmetro ***dwStyle*** quando você criar janelas ***BUTTON***.

Estilo	Descrição
BS_3STATE	Cria um botão que é o mesmo que uma caixa de seleção, exceto que a caixa pode estar em um estado desabilitado, bem como em um estado *assinalado* ou *não-assinalado*. Use o estado desabilitado para mostrar que seu programa não determinou o estado da caixa de seleção.
BS_AUTO3STATE	Cria um botão que é o mesmo que uma caixa de seleção de três botões.
BS_AUTOCHECKBOX	Cria um botão que é o mesmo que uma caixa de seleção, exceto que o estado de seleção muda automaticamente entre *assinalado* e *não assinalado* cada vez que o usuário seleciona a caixa de seleção.
BS_AUTORADIOBUTTON	Cria um botão que é o mesmo que um botão de opção, exceto que, quando o usuário o seleciona, o Windows automaticamente define o estado de seleção do botão para *assinalado*, e automaticamente define o estado de seleção para todos os outros botões no mesmo grupo como *não-assinalado*.
BS CHECKBOX	Cria uma pequena caixa de seleção vazia com texto. Por padrão, seu programa exibe o texto à direita da caixa de seleção. Para exibir o texto à esquerda da caixa de seleção, combine esse estilo com *BS_LEFTTEXT* (ou com o estilo equivalente *BS_RIGHTBUTTON*).
BS_DEFPUSHBUTTON	Cria um botão que se comporta como um botão de estilo *BS_PUSHBUTTON*, mas também tem um borda preta espessa. Se o botão estiver em uma caixa de diálogo, o usuário poderá selecionar o botão pressionando a tecla Enter, mesmo quando o botão não tiver o foco de entrada Este estilo é útil para permitir que o usuário selecione rapidamente a opção mais provável (a padrão).

Tabela 1279.2 Os valores possíveis para o parâmetro ***dwStyle*** quando você criar janelas ***BUTTON***. (*Continuação*)

Estilo	Descrição
BS_GROUPBOX	Cria um retângulo no qual seu programa pode então agrupar outros controles. Seu programa exibe qualquer texto associado com esse estilo no canto superior esquerdo do retângulo. *GROUPBOX* também é comumente conhecido como uma *frame* (uma estrutura).
BS_LEFTTEXT	Coloca o texto no lado esquerdo do botão de opção ou da caixa de seleção quando combinado com um estilo de botão ou de caixa de seleção. O mesmo que o estilo *BS_RIGHTBUTTON*.
BS_OWNERDRAW	Cria um botão desenhado pelo proprietário. A janela proprietária recebe uma mensagem *WM_MEASUREITEM* quando o Windows cria o botão, e recebe uma mensagem *WM_DRAWITEM* quando um aspecto visual do botão muda. Não combine o estilo *BS_OWNERDRAW* com quaisquer outros estilos de botão.
BS_PUSHBUTTON	Cria um botão que envia uma mensagem *WM_COMMAND* para a janela proprietária quando o usuário seleciona o botão.
BS_RADIOBUTTON	Cria um círculo pequeno com o texto. Por padrão, seu programa exibe o texto à direita do círculo. Para exibir o texto à esquerda do círculo, combine este sinalizador com o estilo *BS_LEFTTEXT* (ou com o estilo equivalente *BS_RIGHTBUTTON*). Use botões de opção para grupos de escolhas relacionadas mas mutuamente exclusivas.
BS_USERBUTTON	Este estilo é obsoleto, mas permanece compatível com as versões de 16 bits do Windows. Os aplicativos baseados no Win32 devem usar em lugar dele *BS_OWNERDRAW*.
BS_BITMAP	Especifica que o botão exibe um bitmap.
BS_BOTTOM	Coloca o texto na parte inferior do retângulo do botão.
BS_CENTER,	Centraliza o texto horizontalmente no retângulo do botão.
BS_ICON	Especifica que o botão exibe um ícone.
BS_LEFT	Justifica à esquerda o texto no retângulo do botão. No entanto, se o botão for uma caixa de seleção ou um botão de opção que não tem o estilo *BS_RIGHTBUTTON*, o texto ainda será justificado à esquerda, mas no lado direito da caixa de seleção ou do botão de opção.
BS_MULTILINE	Quebra o texto do botão em múltiplas linhas se a string de texto é grande demais para caber em uma única linha no retângulo do botão.
BS_NOTIFY	Habilita um botão a enviar mensagens de notificação *BN_DBLCLK*, *BN_KILLFOCUS* e *BN_SETFOCUS* à janela-mãe do botão. Observe que os botões enviam a mensagem de notificação *BN_CLICKED* independente se o botão tem ou não o estilo *BS_NOTIFY*.
BS_PUSHLIKE	Faz um botão (tal como uma caixa de seleção, caixa de seleção de três estados ou botão de opção) ter o aspecto e atuar como um botão. O botão parece elevado quando não está assinalado e parece afundado quando está assinalado.
BS_RIGHT	Justifica o texto à direita no retângulo de um botão. No entanto, se o botão for uma caixa de verificação ou um botão de rádio que não tem o estilo BS_RIGHTBUTTON, o texto ainda está justificado à direita, mas no lado direito da caixa de verificação ou do botão de rádio.
BS_RIGHTBUTTON	Posiciona o círculo do botão de opção ou o quadrado da caixa de seleção no lado direito do retângulo do botão. O mesmo que o estilo *BS_LEFTTEXT*.

Tabela 1279.2 Os valores possíveis para o parâmetro ***dwStyle*** quando você criar janelas ***BUTTON***. *(Continuação)*

Estilo	Descrição
BS_TEXT	Especifica que o botão exibe texto.
BS_TOP	Coloca texto no alto do retângulo do botão.
BS_VCENTER	Coloca texto no meio (verticalmente) do retângulo do botão.

Exatamente como a janela *BUTTON* tem seus próprios estilos, assim também faz cada uma das outras classes de janela predefinidas. No entanto, esta dica não as lista, pois o espaço necessário para listar cada estilo seria imenso.

Nota: A maioria dos pacotes de desenvolvimento em C++ para o Windows inclui informações sobre os estilos das janelas na Ajuda on-line. Consulte na Ajuda on-line o tópico CreateWindow para saber mais.

CRIANDO JANELAS COM ESTILOS ESTENDIDOS 1280

Em Dicas anteriores, você aprendeu como usar a função *CreateWindow* para criar janelas personalizadas e predefinidas dentro de seus programas. Além disso, seus programas podem usar a função *CreateWindowEx* para criar janelas com estilos estendidos, tais como sobrepostas, instantâneas ou estilos de janela-filha. Você usará a função *CreateWindowEx* dentro de seus programas como mostrado no protótipo a seguir:

```
HWND CreateWindowEx (
   DWORD dwExEstilo,      // estilo de janela estendido
   LPCTSTR lpNomeClasse,  // ponteiro para nome de classe registrado
   LPCTSTR lpNomeJanela,  // ponteiro para nome de janela
   DWORD dwEstilo,        // estilo da janela
   int x,                 // posição horizontal da janela
   int y,                 // posição vertical da janela
   int nLarg,             // largura da janela
   int nAltura,           // altura da janela
   HWND hWnd mãe,         // indicativo para janela-mãe ou janela proprietária
   HMENU hMenu,           // indicativo para menu, ou janela-filha
   HINSTANCE hCopia,      // indicativo para ocorrência do aplicativo
   LPVOID lpParam         // ponteiro para dados da criação da janela
);
```

Como você pode ver, a função *CreateWindowEx* aceita os mesmos parâmetros que a função *CreateWindow*, com exceção do parâmetro *dwExEstilo*, que especifica o estilo estendido da janela. Observe que seus programas podem especificar um número ilimitado de *dwExEstilo* para as suas janelas, desde que você vincule os estilos com uma operação *OU* bit a bit. A Tabela 1280 lista os valores possíveis para o parâmetro *dwExEstilo*.

Tabela 1280 Os valores possíveis para o parâmetro ***dwExEstilo***.

Estilo	Descrição
WS_EX_ACCEPTFILES	Especifica que uma janela que o sistema operacional cria com esse estilo aceita que os arquivos sejam arrastados e soltos.
WS_EX_APPWINDOW	Força uma janela de alto nível a ir para a barra de tarefas quando o Windows minimiza a janela.
WS_EX_CLIENTEDGE	Especifica que uma janela tem uma borda com uma ponta afundada.

Tabela 1280 Os valores possíveis para o parâmetro *dwExEstilo*. (*Continuação*)

Estilo	Descrição
WS_EX_CONTEXTHELP	Inclui uma interrogação na barra de título da janela. Quando o usuário der um clique com o mouse no ponto de interrogação, o cursor mudará para uma interrogação com um ponteiro. Se o usuário então der um clique com o mouse em uma janela-filha, a filha receberá uma mensagem *WM_HELP*. A janela-filha deverá passar a mensagem para o procedimento de janela da janela-mãe, que deverá chamar a função *WinHelp* usando o comando *HELP WM HELP*. O aplicativo de ajuda exibe uma janela que tipicamente contém ajuda para a janela-filha. Seus programas não podem usar *WS_EX_CONTEXTHELP* com os estilos *WS_MAXIMIZEBOX* e *WS_MINIMIZEBOX*.
WS EX_CONTROLPARENT	Permite que o usuário navegue entre as janelas-filha da janela usando a tecla Tab.
WM_EX_DLGMODALFRAME	Cria uma janela que tem uma borda dupla. Você tem a opção de criar a janela com uma barra de título especificando o estilo *WS_CAPTION* no parâmetro *dwStyle*.
WS_EX_LEFT	O Windows tem propriedades genéricas alinhadas à esquerda. Este é o estilo padrão.
WS_EX_LEFTSCROLLBAR	Coloque a barra de rolagem vertical (se presente) à esquerda da área cliente se o idioma do núcleo for hebraico, árabe ou outro idioma que suporte o alinhamento na ordem da leitura. Para outros idiomas, o Windows ignora o estilo e não o trata como um erro.
WS_EX_LTRREADING	Exibe o texto da janela usando as propriedades de leitura Left to Right. Esse é o estilo padrão.
WS_EX_MDICHILD	Cria uma janela-filha da Interface de Múltiplos Documentos (MDI).
WS_EX_NOPARENTNOTIFY	Especifica que uma janela-filha criada com esse estilo não enviará a mensagem *WM_PARENTNOTIFY* para sua janela-filha quando o Windows criar ou destruir
WS_EX_OVERLAPPEDWINDOW	Combina os estilos *WS_EX_CLIENTEDGE* e *WS_EX_WINDOWEDGE*.
WS_EX_PALETTEWINDOW	Combina os estilos *WS_EX_WINDOWEDGE*, *WS_EX_TOOLWINDOW* e *WS_EX_TOPMOST*.
WS EX RIGHT	O Windows tem propriedade genérica de alinhamento à direita, embora a propriedade seja dependente da classe da janela. Esse estilo terá efeito somente se o idioma da interface for hebraico, árabe ou outro idioma que suporte o alinhamento da ordem de leitura; caso contrário, o Windows ignorará o estilo e não o tratará como um erro. Usando o estilo *WS_EX_RIGHT* para controles estáticos ou de edição terá o mesmo efeito que usar os estilos *BS_RIGHT* e *BS_RIGHTBUTTON*.
WS_EX_RIGHTSCROLLBAR	Coloca a barra de rolagem vertical (se presente) à direita da área cliente. Este é o estilo padrão.
WS_EX_RTLREADING	Exibe o texto da janela usando as propriedades de ordem de leitura Right to Left se o idioma da interface for hebraico, árabe ou outro idioma que suporte o alinhamento da ordem de leitura. Para outros idiomas, o Windows ignora o estilo e não o trata como um erro.
WS_EX STATICEDGE	Cria uma janela com um estilo de borda tridimensional destinado para o Windows usar com itens que não aceitam entrada do usuário.

Tabela 1280 Os valores possíveis para o parâmetro ***dwExEstilo***. (*Continuação*)

Estilo	Descrição
WS_EX_TOOLWINDOW	Cria uma janela de ferramenta, que é uma janela que seu programa quer que o Windows use como uma barra de ferramentas flutuante. Uma janela de ferramenta tem uma barra de título que é mais curta que uma barra de título normal, e o Windows usará uma fonte menor para desenhar o título da janela. Uma janela de ferramenta não aparece na barra de tarefa ou na caixa de diálogo que aparece quando o usuário pressiona Alt+Tab. Se uma janela de ferramenta tiver um menu do sistema, o Windows não exibirá seu ícone na barra de título. No entanto, você pode dar um clique no botão direito do mouse ou pressionar Alt+Espaço para exibir o menu do sistema.
WS_EX_TOPMOST	Especifica que o Windows deve colocar uma janela, criada com esse estilo, acima de todas as janelas que não são de nível mais alto e que a janela deve ficar acima delas, mesmo quando seu programa desativa a janela. Para acrescentar ou remover esse estilo, use a função *SetWindowPos*.
WS_EX_TRANSPARENT	Especifica que o Windows precisa criar a janela como transparente. Isto é, essa janela não obscurece quaisquer janelas que estão abaixo dela. Uma janela criada com esse estilo recebe mensagens *WM_PAINT* somente após seu programa ter atualizado todas as janelas-irmã abaixo dela.
WS_EX_WINDOWEDGE	Especifica que uma janela tem uma borda com uma ponta elevada.

À medida que seus programas se tornarem mais complexos, com mais e mais janelas dentro deles, provavelmente você se verá usando *CreateWindowEx* com mais freqüência do que *CreateWindow*. Por exemplo, o comando a seguir cria uma janela com um pequeno título (tal como uma janela de caixa de ferramentas exibe) em vez de um título em tamanho completo:

```
hJanPrin = CreateWindowEx(WS_EX+SMCAPTION, lpszNomeAplic,
             lpszTitulo, WS_OVERLAPPEDWINDOW |
             WS_CLIPCHILDREN, CW_USEDEFAULT, 0,
             CW_USEDEFAULT, 0, NULL, NULL, hCopia, NULL);
```

DESTRUINDO JANELAS 1281

Em todas as dicas anteriores você aprendeu como seus programas criarão janelas personalizadas, janelas predefinidas e janelas com estilos estendidos. Como você sabe, cada ocorrência de janela é um objeto de uma *classe de janela*, e, como aprendeu anteriormente, seus programas sempre devem destruir os objetos após o programa completar seu processamento no objeto. No entanto, como você não usa *new*, *malloc* ou uma função similar para alocar memória para a janela quando seu programa cria a janela, você não pode usar *delete* ou outra função de liberação de memória para liberar essa memória e destruir a janela. Em vez disso, você precisa usar a chamada *DestroyWindow* da API. *DestroyWindow* removerá a janela que você passar para ela dentro de seu único parâmetro. Você usará a função *DestroyWindow* dentro de seus programas como mostrado no protótipo a seguir:

```
BOOL DestroyWindow(HWND hJan);
```

DestroyWindow retorna *True* se for bem-sucedida, e *False* em caso contrário. Antes de destruir a janela, *DestroyWindow* envia as mensagens *WM_DESTROY* e *WM_NCDESTROY* para a janela para desativá-la. Seu procedimento de janela deve responder às mensagens *WM_DESTROY* e *WM_NCDESTROY* antes que o programa destrua a janela. *DestroyWindow* destrói o menu da janela, e esvazia a fila de mensagens do encadeamento.

Além de usar *DestroyWindow* para destruir janelas que seus programas criam usando *CreateWindow*, seus programas também podem usar *DestroyWindow* para destruir os diálogos sem modo que seus programas criam usando *CreateDialog*. No entanto, você tipicamente usará *DestroyWindow* dentro de uma rotina *ProcJan* para responder a um comando de saída de dentro do programa. Por exemplo, o fragmento de código a seguir mostra como os programas *generico.cpp* e *exibe_3.cpp* usam a opção Sair do menu Arquivo para destruir a janela do programa:

```
case IDM_SAIR :
   DestroyWindow(hJan);
    break;
```

1282 COMPREENDENDO A FUNÇÃO DA API REGISTERCLASSEX

Na Dica 1269, você aprendeu que seus programas usarão a função *RegisterClass* da API para registrar uma classe personalizada. A função *RegisterClass* difere de *RegisterClass*, pois lhe permite registrar uma classe com um pequeno ícone que o Windows colocará na barra de título de todas as ocorrências das classes registradas. Você usará a função *RegisterClassEx* dentro de seus programas, como mostrado no seguinte protótipo:

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);
```

A função *RegisterClassEx* retorna um *ATOM*, exatamente como a função *RegisterClass*. No entanto, *RegisterClassEx* recebe um único parâmetro, um ponteiro para uma estrutura *WNDCLASSEX* (ao contrário da estrutura *WNDCLASS* que *RegisterClass* recebe). Você precisa preencher a estrutura com os atributos de classe apropriados antes de passá-la para a função. A estrutura *WNDCLASSEX* é ligeiramente diferente da estrutura *WNDCLASS*, como mostrado aqui:

```
typedef struct tagWNDCLASSEX
 {
   UINT        style;
   WNDPROC     lpfnWndProc;
   int         cbClsExtra;
   int         cbWndExtra;
   HINSTANCE   hInstance;
   HICON       hIcon;
   HCURSOR     hCursor;
   HBRUSH   hbrBackground;
   LPCTSTR     lpszMenuName;
   LPCSTR   lpszClassName;
   HICON       hIconSm;
 }  WNDCLASSEX;
```

A única adição que *WNDCLASSEX* faz na estrutura *WNDCLASS* é acrescentar o indicativo *HICON* como o último item. Quando você inicializar a estrutura *WNDCLASSEX*, geralmente usará o comando *LoadIcon* para atribuir um valor ao membro *hIconSm*.

Nota: Sob o Windows 95, RegisterClassEx falha se o membro cbWndExtra ou o membro cbClsExtra da estrutura WNDCLASSEX contém mais do que 40 bytes.

1283 ANEXANDO INFORMAÇÕES EM UMA JANELA COM SETPROP

Criar janelas dentro de seus programas é um processo relativamente simples. Além de criar janelas, seus programas podem anexar uma lista de dados associados, conhecidos como *itens de propriedade*, em uma janela. Geralmente, você anexará itens de propriedades a uma janela para ajudar seus programas a manter informações sem o uso de variáveis globais. Qualquer seção dentro do programa que conheça ou que possa acessar o indicativo para uma determinada janela pode então obter as informações que você anexou anteriormente à janela dentro da lista de itens de propriedade. Dentro de seus programas, você usará a função *SetProp* para acrescentar um novo item ou alterar um item existente na lista de propriedade da janela especificada. A função *SetProp* acrescenta um novo item na lista se a string de caracteres especificada ainda não existir na lista. A nova entrada contém a string e o indicativo. Caso contrário, a função substitui o indicativo atual da string com o indicativo especificado. Você chamará a função *SetProp* dentro de seus programas como segue:

```
BOOL SetProp (
   HWND hJan           // indicativo da janela
   LPCTSTR lpString;   // átomo ou endereço da string
   HANDLE hDados;      // indicativo dos dados
);
```

A função *SetProp* retornará *True* se for bem-sucedida, e *False* em caso contrário. A Tabela 1283 lista os parâmetros e suas descrições para a função *SetProp*.

Tabela 1283 Os parâmetros para a função *SetProp*.

Parâmetro	Descrição
hJan	Identifica a janela cuja lista de propriedade recebe a nova entrada.
lpString	Aponta para uma string terminada por *NULL* ou contém um *ATOM* que identifica uma string. Se esse parâmetro for um *ATOM*, ele precisará ser um *ATOM* global que uma chamada anterior à função *GlobalAddAtom* criou. Seu programa precisa colocar o *ATOM*, um valor de 16 bits, na palavra menos significativa de *lpString*. A palavra mais significativa precisa ser zero.
hDados	Identifica dados para a função *SetProp* para copiar para a lista de propriedade. O parâmetro *hData* pode identificar qualquer valor útil para o aplicativo.

Antes de destruir uma janela (isto é, antes de processar a mensagem *WM_DESTROY*), um aplicativo precisa remover todos os itens que tenha acrescentado na lista de propriedades. O aplicativo precisa usar a função *RemoveProp* para remover as entradas. O CD-ROM que acompanha este livro inclui o programa *exi_prop.cpp*, que define uma propriedade para uma janela na sua criação, depois exibe a propriedade quando o usuário seleciona a opção *Testar!* Como no programa anterior, a única diferença significativa entre o arquivo *generico.cpp* e *prop.cpp* está dentro da função *ProcJan*.

O código dentro da função *ProcJan* para o programa *exi-prop* efetua várias novas atividades. Primeiro, a função *Procjan* acrescenta um comando *case* para verificar a chegada de uma mensagem *WM_CREATE*. Quando o programa receber a mensagem *WM-CREATE*, ele acrescentará uma propriedade com o valor de string *Valor para a Propriedade* à lista de propriedade da janela. Quando você selecionar a opção *Testar!* mais tarde, o programa exibirá essa única propriedade conhecida dentro de uma *Caixa de Mensagem*. Você aprenderá sobre as Caixas de Mensagem na Dica 1286. A Dica 1284 detalha como você pode listar as propriedades de uma janela, quando o programa não conhecer a chave real da propriedade.

USANDO ENUMPROPS PARA LISTAR AS PROPRIEDADES DE UMA JANELA 1284

Na dica anterior, você aprendeu como usar a função *SetProp* para definir uma propriedade de janela e, brevemente, como usar a função *GetProp* para retornar essa propriedade em alguma outra parte dentro de seus programas. Como você viu, a função *GetProp* requer que seu programa passe um ponteiro para uma string terminada por *NULL* ou um *ATOM* que identifica a propriedade que você quer recuperar. Algumas vezes dentro de seus programas você precisará recuperar todas as propriedades associadas com uma janela sem conhecer os nomes das propriedades, quantas propriedades existem, e assim por diante. A função *EnumProps* lhe permite enumerar (listar) todos os itens na lista de propriedade de uma janela passando cada item, um de cada vez, para a função de callback especificada. *EnumProps* continua até que enumere o último item ou a função de callback retorne *False*. Dentro de seus programas, você chamará a função *EnumProps* com a seguinte forma geral:

```
int EnumProps  (
   HWND hJan;                  // indicativo para a janela
   PROPENUMPROC lpEnumFunc     // ponteiro para a função de callback
);
```

Antes da chamada da função *EnumProps*, você precisa definir a função de callback que *EnumProps* chamará. O formato geral dessa função é mostrado aqui (observe que você pode nomear a função do modo como quiser; o cabeçalho *PropEnumProc* é simplesmente um marcador de lugar):

```
BOOL CALLBACK   PropEnumProc (
   HWND hJan,              // indicativo da janela
   LPCTSTR lpszString,     // componente string da propriedade
   HANDLE  hDados          // componente de indicativo de dados da propriedade
);
```

As restrições a seguir aplicam-se à função de callback *PropEnumProc:*

1. A função de callback não pode ceder o controle ou fazer qualquer coisa que possa ceder o controle para outras tarefas.
2. A função de callback pode chamar a função *RemoveProp*. No entanto, *RemoveProp* pode remover somente a propriedade passada para a função de callback por meio dos parâmetros da função de callback.
3. A função de callback não deve tentar adicionar propriedades.

Quando você usa as funções *EnumProps* e *PropEnumProc* juntas, seus programas podem listar cada propriedade anteriormente associada com uma janela na ordem em que seu programa associou as propriedades com a janela. Por exemplo, o CD-ROM que acompanha este livro inclui o programa *EnumProps.cpp*, que modifica um pouco o programa *ProcJan* e acrescenta uma nova função, *EnumPropsProc*, no código de *generico.cpp*. Para fazer a função *EnumProps* trabalhar corretamente, apague a função *ProcJan* existente do programa *generico.cpp* e, em seu lugar, acrescente o código a seguir:

```
BOOL CALLBACK EnumPropsProc(HWND hJan, LPCTSTR lpszString, HANDLE hData)
 {
    MessageBox( hJan, (LPCTSTR)hDado, lpszString, MB_OK );
    return( TRUE );
 }

LRESULT CALLBACK ProcJan(HWND hJan, UINT uMsg, WPARAM wParam, LPARAM lParam)
 {
static LPCTSTR szProp1 = "Valor para a Propriedade 1";
static LPCTSTR szProp2 = "Valor para a Propriedade 2";
static LPCTSTR szProp3 = "Valor para a Propriedade 3";

   switch(uMsg)
   {
      case WM_CREATE :
         // Acrescenta itens de propriedade que contêm
         // ponteiros string para a janela principal
         SetProp( hJan,"Propriedade 1",(HANDLE)szProp1 );
         SetProp( hJan,"Propriedade 2",(HANDLE)szProp2 );
         SetProp( hJan,"Propriedade 3",(HANDLE)szProp3 );
         break;

      case WM_COMMAND :
         switch(LOWORD( wParam ))
         {
            case IDM_TESTAR :
                // Enumera as propriedades e exibe em uma
                // caixa de mensagem o nome e o valor da
                // propriedade
                EnumProps( hJan, (PROPENUMPROC) EnumPropsProc );
                      break;

               case IDM_SOBRE :
                      DialogBox(hCop, "CxSobre", hJan, (DLGPROC)Sobre);
                      break;

               case IDM_SAIR :
                      DestroyWindow(hJan);
                      break;
            }
            break;

      case WM_DESTROY :
            PostQuitMessage(0);
            break;
```

```
        default :
                return(DefWindowProc(hJan, uMsg, wParam, lParam));
    }

    return( 0L );
}
```

Quando você copilar e executar o programa *EnumProps* e selecionar a opção *Testar*, o programa exibirá três caixas de mensagem em seqüência, cada uma das quais exibe a string "Valor para a Propriedade *n*", onde *n* é igual a 1, 2 ou 3, dependendo da caixa de mensagem. Observe a linha que chama a função *EnumProps:*

```
EnumProps(hJan, (PROPENUMPROC)EnumPropsProc);
```

Lembre-se, o operador (*PROPENUMPROC*) é uma conversão explícita para um ponteiro do tipo *PROPENUMPROC*. Esse ponteiro então aponta para o local dentro da memória onde a função *EnumPropsProc* inicia.

COMPREENDENDO AS FUNÇÕES DE CALLBACK 1285

Em dicas anteriores, você usou funções de *callback* para obter vários objetivos. Como você viu, a função de *callback* é crítica para o processamento do Windows. Para compreender logicamente como a função de *callback* se adequa dentro de uma chamada de função, considere o diagrama lógico mostrado na Figura 1285.

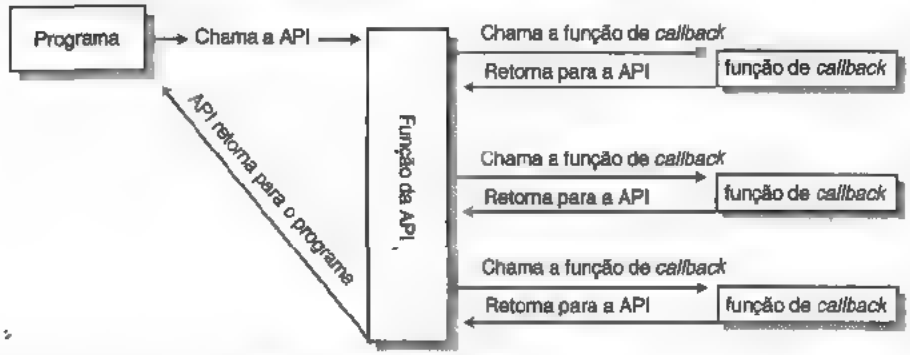

Figura 1285 O modelo lógico do processamento de uma função de callback.

Seus programas usarão funções de *callback* de muitos modos; no entanto, como já detalhado, você mais freqüentemente usará as funções de *callback* com chamadas da API que geram um número desconhecido de valores de retorno. Com frequência você encontrará funções que executam atividades de *callback* cujos nomes iniciam com *Enum*, incluindo *EnumFontFamilies*, *EnumWindows*, *EnumProps*, e assim por diante.

COMPREENDENDO A FUNÇÃO MESSAGEBOX 1286

Nas Dicas 1283 e 1284, seus programas usaram a função *MessageBox* para exibir caixas de diálogo simples que requeriam interação do usuário antes de o programa continuar seu processamento. Em dicas posteriores você aprenderá sobre a criação de caixas de diálogo de muitos tipos diferentes; no entanto para as caixas de diálogo simples de entrada do usuário, seus programas podem usar a função *MessageBox*. A função *MessageBox* cria, exibe e opera uma caixa de mensagem. A caixa de mensagem contém uma mensagem e um título definido pelo aplicativo, mais qualquer combinação de ícones e botões predefinidos. Uma caixa de mensagem não pode exibir outras janelas dentro de si mesma. Seus programas usarão a seguinte forma generalizada da função *MessageBox* para exibir caixas de mensagem:

```
int MessageBox (
    HWND hJan,        // indicativo da janela proprietária
    LPCTSTR lpTexto,     // endereço do texto na caixa de msg
    LPCTSTR lpTitulo,    // endereço do título da caixa de mensagem
    UINT uTipo           // estilo da caixa de mensagem
);
```

A função *MessageBox* aceita os parâmetros detalhados na Tabela 1286.1

Tabela 1286.1 Os parâmetros para a função ***MessageBox***

Sinalizador	Descrição
hJan	Identifica a janela proprietária da caixa de mensagem que você quer criar. Se esse parâmetro for *NULL*, a caixa de mensagem não terá janela proprietária.
lpTexto	Aponta para uma string terminada por *NULL* que contém a mensagem que a caixa de mensagem precisa exibir.
lpTitulo	Aponta para uma string terminada por *NULL* usada para o título da caixa de diálogo. Se esse parâmetro for *NULL*, seu programa usará o título padrão *Erro*.
uTipo	Especifica um conjunto de bits sinalizadores que determinam o conteúdo e o comportamento da caixa de diálogo. Esse parâmetro pode ser uma combinação de sinalizadores a partir do seguinte grupo de sinalizadores. A Tabela 1286.2 detalha os sinalizadores que seu programa usará para indicar os botões que uma caixa de mensagem contém.

Além do texto que a caixa de mensagem exibe (que o parâmetro *lpTexto* define) e o título para a caixa de mensagem (que o parâmetro *lpTitulo* define), seus programas mais frequentemente manipularão o parâmetro *uTipo* para alterar os botões e quaisquer ícones que você quer que apareçam na caixa de mensagem. A Tabela 1286.2 lista os valores possíveis que você pode usar para o parâmetro *uTipo* para controlar o número de botões que aparece na caixa de mensagem.

Tabela 1286.2 Os valores possíveis para o parâmetro ***uTipo*** para controlar o número de botões.

Sinalizador	Significado
MB_ABORTRETRYIGNORE	A caixa de mensagem contém três botões *Abortar*, *Repetir* e *Ignorar*.
MB_OK	A caixa de mensagem contém um botão: *OK*. O sinalizador *MB_OK* é o valor padrão para o parâmetro *uTipo*.
MB_OKCANCEL	A caixa de mensagem contém dois botões: *OK* e *Cancelar*.
MB_RETRYCANCEL	A caixa de mensagem contém dois botões: *Repetir* e *Cancelar*.
MB_YESNO	A caixa de mensagem contém dois botões: *Sim* e *Não*.
MB_YESNOCANCEL	A caixa de mensagem contém três botões *Sim*, *Não* e *Cancelar*

Além de controlar o número e o título dos botões que aparecem dentro da caixa de mensagem, seus programas também podem controlar se um ícone aparece ou não dentro da caixa de mensagem. A definição padrão é *MB_NOICON*. A Tabela 1286.3 lista os valores possíveis para o parâmetro *uTipo* para controlar a aparência do ícone dentro da caixa.

Tabela 1286.3 Os valores possíveis para o parâmetro ***uTipo*** para controlar o ícone da caixa de mensagem.

Sinalizador	Significado
MB_ICONEXCLAMATION	Um ícone de ponto de exclamação aparece na caixa de mensagem.
MB_ICONWARNING	Um ícone de ponto de exclamação aparece na caixa de mensagem.
MB_ICONINFORMATION	Um ícone que consiste de uma letra minúscula *i* em um círculo que aparece na caixa de mensagem.
MB1CONASTERISK	Um ícone que consiste de uma letra minúscula *i* em um círculo aparece na caixa de mensagem.
MB ICONQUESTION	Um ícone de ponto de interrogação aparece na caixa de mensagem.

Tabela 1286.3 Os valores possíveis para o parâmetro ***uTipo*** para controlar o ícone da caixa de mensagem. (*Continuação*)

Sinalizador	Significado
MB_ICONSTOP	Um ícone de sinal de parar aparece na caixa de mensagem.
MB_ICONERROR	Um ícone de sinal de parar aparece na caixa de mensagem.
MB_ICONHAND	Um ícone de sinal de parar aparece na caixa de mensagem
MB_NOICON	Nenhum ícone aparece na caixa de mensagem.

A função *MessageBox* suporta dezenas de constantes adicionais para o parâmetro *uTipo*. No entanto, as constantes listadas nas Tabelas 1286.2 e 1286.3 são as mais comumente usadas. Para saber quais são as constantes adicionais, consulte a documentação on-line do seu compilador.

O CD-ROM que acompanha este livro inclui o programa *ExibeMsg.cpp*, que gera um aviso sonoro, sobre o qual você aprenderá na dica a seguir, depois exibe "*Olá!*" e aguarda a digitação de uma tecla. O programa não executa processamento útil da seleção que o usuário faz dentro da caixa de mensagem, mas poderia fazer isso com facilidade.

COMPREENDENDO A FUNÇÃO MESSAGEBEEP 1287

Na dica anterior, você usou a função *MessageBox* para gerar uma caixa de diálogo simples. O programa *ExibeMsg* também chamou a função *MessageBeep* quando criou a caixa de mensagem. A função *MessageBeep* reproduz uma forma de onda sonora. Uma entrada na seção [*sounds*] do registro do sistema identifica a forma de onda para cada tipo de som. Seus programas usarão a função *MessageBeep* de acordo com a seguinte forma geral:

```
BOOL MessageBeep(UINT uTipo)
```

O parâmetro *uTipo* especifica o tipo do som, como identificado por uma entrada na seção [*sounds*] do registro. Você poderá reproduzir os sons a partir de dentro do Painel de Controle do Windows se não tiver certeza do som que produzem. O parâmetro de som *uTipo* pode ser um dos valores listados na Tabela 1287.

Tabela 1287 Os valores possíveis para o parâmetro de som ***uTipo***.

Valor	Som
0xFFFFFFFF	Aviso sonoro padrão usando o alto-falante do computador
MB_ICONASTERISK	Asterisco do sistema
MB_ICONEXCLAMATION	Exclamação do sistema
MB_ICONHAND	Mão do sistema
MB_ICONQUESTION	Pergunta do sistema
MB_OK	Padrão do sistema

Seus programas deverão usar a função *MessageBeep* para criar sons simples, os quais você tipicamente usará para alertar o usuário de algum evento.

REVISITANDO AS MENSAGENS 1288

Como você aprendeu, o tratamento de mensagem está no "coração" daquilo que faz com que os aplicativos Windows trabalhem. Tanto o sistema operacional quanto os aplicativos que rodam sob ele geram mensagens para todo evento que ocorre no Windows. As mensagens são fundamentalmente importantes para o valor do Windows como um sistema operacional de multitarefas. Como você aprenderá, cada tarefa (ou programa) usa um ou mais encadeamentos dentro do sistema operacional. As plataformas de 32 bits do Windows (Windows NT e Windows 95) mantêm um conjunto separado de mensagens (uma *fila de mensagem*) para cada encadeamento que estiver em execução no sistema operacional.

O Windows gera mensagens para todo evento de hardware que ocorre, tal como a digitação de uma tecla ou um clique no mouse do usuário. O Windows então passa cada mensagem à fila de mensagens apropriada. Em outras palavras, se o usuário der um clique no mouse mas não dentro do seu aplicativo, seu aplicativo não saberá

que o usuário deu um clique no mouse. Ocasionalmente, o sistema gerará várias cópias de uma mensagem que ele coloca ao mesmo tempo em múltiplas filas de mensagem. A Figura 1288 mostra um diagrama simples de como o Windows processa mensagens em várias filas de mensagem.

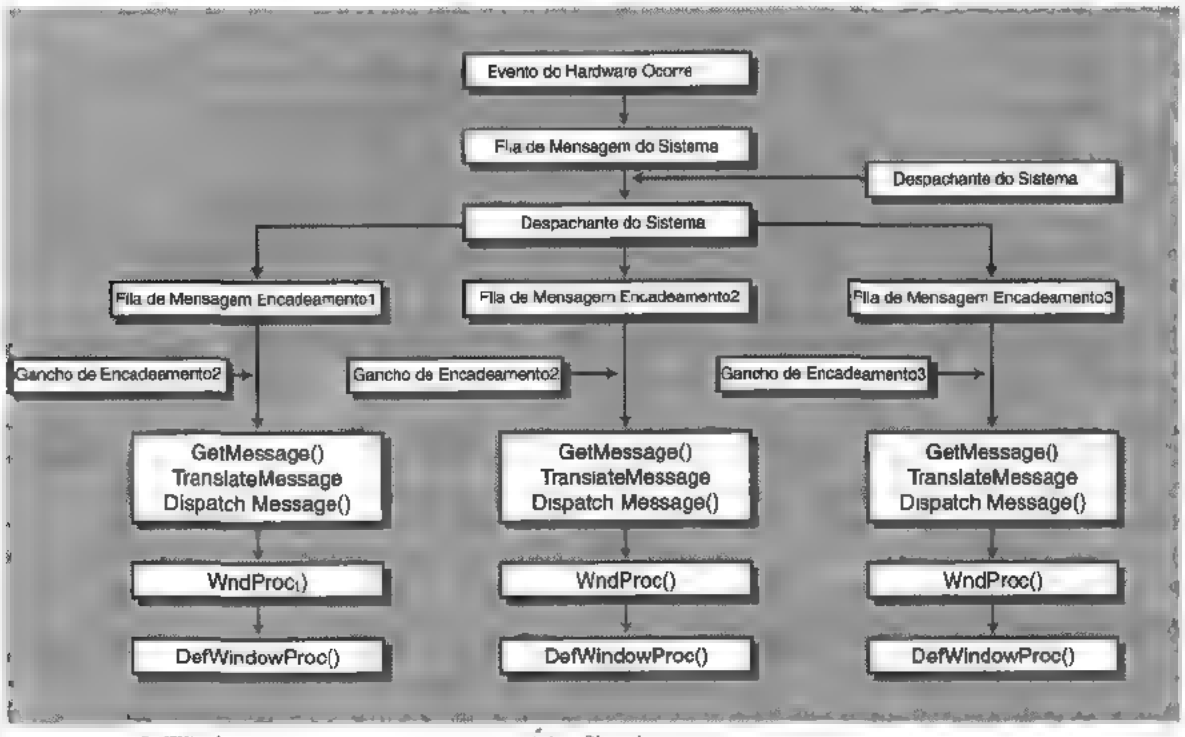

Figura 1288 O Windows processa mensagens em várias filas de mensagens.

1289 · COMPREENDENDO O FLUXO DE MENSAGENS

Como você aprendeu, a estrutura de mensagem do Windows é fundamental para o modo como o Windows gerencia múltiplas tarefas em sequência. Você deve compreender o fluxo de mensagens não apenas para a fila de mensagens do programa, mas também a partir da fila de mensagem do programa no laço de mensagem do programa. Quando o Windows aceitar uma mensagem a partir do hardware do computador, ele determinará internamente para quais filas de mensagens ele passará essa mensagem. Após o Windows passar a mensagem na fila de mensagens do programa, o programa processará cada mensagem uma de cada vez. Por exemplo, algumas vezes, ao estar digitando em um processador de textos, você digitará mais rápido do que a tela pode exibir as letras. No entanto, o programa poderá manter sua digitação, mesmo enquanto a tela tenta pegar, pois o Windows está armazenando cada tecla que você coloca na fila de mensagens do programa, como mostrado na Figura 1289.1.

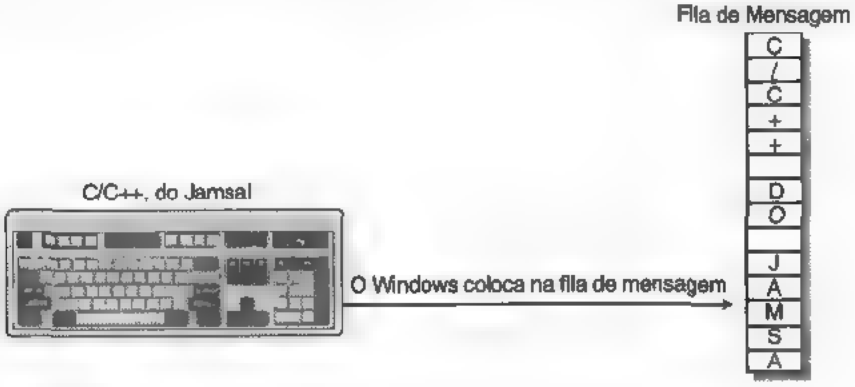

Figura 1289.1 O Windows processa as teclas e coloca-as na fila de mensagem

Após o Windows colocar as teclas dentro da fila de mensagem, o programa pega as mensagens uma de cada vez da fila de mensagem, recuperando a mensagem mais antiga primeiro e continuando em ordem até que recupere a última mensagem da fila. Após recuperar cada mensagem, o programa usa o laço de mensagem para chamar a função de mensagem de callback do programa, que processa cada uma das entradas, como mostrado na Figura 1289.2.

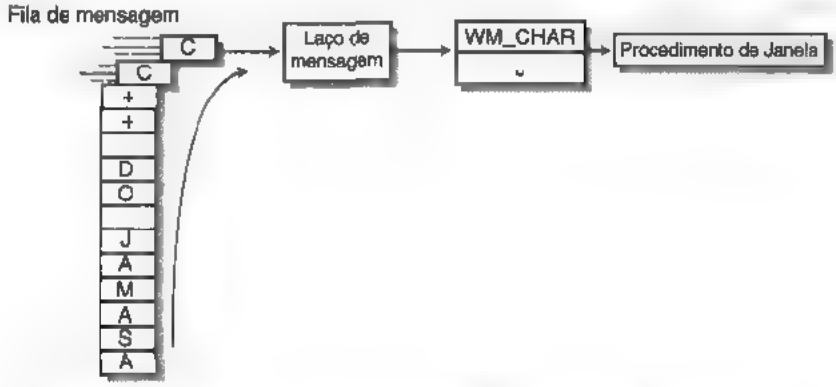

Figura 1289.2 O laço de mensagem chama a função de *callback* de mensagem do programa.

Como você aprendeu, seus programas então verificarão os valores dentro das mensagens para determinar como responder a elas. Se, por exemplo, o comando for uma tecla que o usuário quer colocar dentro do documento real do processador de texto, o laço de mensagem então despachará o caractere para a janela atual e acrescentará o caractere ao documento de processamento de texto em sua posição atual, como mostrado na Figura 1289.3.

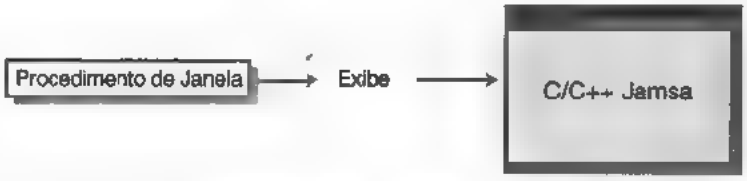

Figura 1289.3 A função de *callback* de mensagem insere o caractere no documento do processamento de texto.

COMPREENDENDO MELHOR OS COMPONENTES DA ESTRUTURA MSG 1290

Como você aprendeu, a mensagem do Windows é um dos blocos de construção fundamentais com base em que você criará programas Windows. Você também aprendeu que o Windows define a estrutura MSG, como mostrado aqui:

```
typedef struct  tagMSG  {
   HWND     hwnd;        // indicativo da janela
   UINT     message;     // ident. da mensagem
   WPARAM   wParam;      // valor de wParam
   LPARAM   lParam;      // valor de lParam
   DWORD    time;        // milissegundos desde o início
   POINT    pt;          // coordenadas da tela da posição atual do mouse
};
```

Embora cada um dos componentes da estrutura da mensagem seja importante por diferentes razões, os dois componentes que você usará mais comumente dentro deste livro são os valores *mensagem* (que *ProcJan* recebe com *uMsg*) e *wParam*. Quando você chamar a função *DispatchMessage*, como seus laços de mensagem sempre chamarão, a função *DispatchMessage* enviará a mensagem para a função *ProcJan*, cujo cabeçalho você definirá como mostrado aqui:

```
LRESULT CALLBACK ProcJan(HWND hJan, UINT uMsg,
                         WPARAM wParam, LPARAM lParam);
```

A função *DispatchMessage* transmite somente os membros *hJan, mensagem, wParam* e *lParam* da estrutura *MSG* para a função de *callback*. Como você aprendeu, a função de *callback* então usa o parâmetro *uMsg* para fazer sua determinação inicial do tipo da mensagem. Se o tipo da mensagem for um comando do Windows, sem parâmetros, a função de *callback* executará seu processamento apropriadamente (por exemplo, *WM_DESTROY*). Por outro lado, se a mensagem for um comando *WM_COMMAND*, a função de *callback* precisará então verificar a palavra menos significativa (os 16 bits menos significativos) do parâmetro *wParam* para determinar qual comando específico o sistema recebeu.

Como já visto, você definirá identificadores de constante para as opções com base em comandos dentro de seus programas para que sua função de *callback* possa processar corretamente os comandos quando ela os receber. Por exemplo, a maioria dos programas que você criou até aqui ou o identificador *IDM_TESTAR* para determinar quando um usuário seleciona a opção de menu *Testar!*. À medida que você continuar a expandir seus programas, verá que usa identificadores com praticamente todos os membros componentes de uma janela para garantir que sua função de *callback* possa processar suas seleções corretamente.

1291 COMPREENDENDO A FUNÇÃO PEEKMESSAGE

Como você aprendeu, seus programas podem recuperar mensagens a partir da fila de mensagem usando a função *GetMessage* ou a função *PeekMessage*. A função *PeekMessage* verifica se há uma mensagem na fila de mensagem de um encadeamento e coloca essa mensagem (se existente) na estrutura especificada, como mostrado aqui:

```
BOOL PeekMessage  (
   LPMSG lpMsg,           // ponteiro de estrutura para uma mensagem
   HWND  hJan,            // indicativo para janela
   UINT  wMsgFiltroMin,   // primeira mensagem
   UINT  wMsgFiltroMax,   // última mensagem
   UINT  wRemoveMsg       // sinalizador de remoção
);
```

A função *PeekMessage* aceita os parâmetros detalhados na Tabela 1291.1.

Tabela 1291.1 Os parâmetros para a função ***PeekMessage***.

Parâmetros	Descrição
lpMsg	Aponta para uma estrutura *MSG* que contém informações de mensagem da fila de aplicativos baseados no Windows.
hJan	Identifica a janela com mensagens que seu programa precisa examinar.
wMsgFiltroMin	Especifica o valor da primeira mensagem no intervalo de mensagens que seu programa precisa examinar.
wMsgFiltroMax	Especifica o valor da última mensagem no intervalo de mensagens que seu programa precisa examinar.
wRemoveMsg	Especifica como seu programa trata as mensagens. Esse parâmetro pode ser um dos valores especificados na Tabela 1291.2.

A Tabela 1291.2 detalha os possíveis valores para o parâmetro ***wRemoveMsg*** dentro da função *PeekMessage*.

Tabela 1291.2 Os possíveis valores para o parâmetro *wRemoveMsg*.

Valor	Descrição
PM_NOREMOVE	Não remove as mensagens da fila após *PeekMessage* completar seu processamento.
PM_REMOVE	Remove as mensagens da fila após *PeekMessage* completar seu processamento.

Você tem a opção de combinar o valor *PM_NOYIELD* com *PM_NOREMOVE* ou *PM_REMOVE*. No entanto, *PM_NOYIELD* não tem efeito nos aplicativos Windows de 32 bits. Ele é definido no Win32 unicamente para oferecer compatibilidade com aplicativos escritos para as versões anteriores do Windows, em que o

Windows o usava para impedir que a tarefa atual parasse e cedesse recursos do sistema a outra tarefa. Os aplicativos Windows de 32 bits sempre rodam simultaneamente.

Ao contrário da função *GetMessage*, a função *PeekMessage* não espera que o Windows coloque uma mensagem na fila antes de retornar para o local de chamada. *PeekMessage* somente recupera mensagens associadas com a janela que o parâmetro *hJan*, ou qualquer um de seus filhos, identifica como a função *IsChild* especifica, e dentro do intervalo de valores de mensagens que os parâmetros *uMsgFiltroMin* e *uMsgFiltroMax* especificam. Se *hJan* for *NULL*, *PeekMessage* recuperará mensagens para qualquer janela que pertença ao encadeamento atual que faz a chamada. (*PeekMessage* não recupera mensagens para as janelas que pertencem a outros encadeamentos). Se *hJan* for -1, *PeekMessage* somente retornará mensagens com um valor *hJan* de *NULL*. Se *wMsgFiltroMin* e *wMsgFiltroMax* forem ambos zero, *PeekMessage* retornará todas as mensagens disponíveis (isto é, ela não efetua quaisquer filtragens de intervalo).

Você pode usar as constantes *WM_KEYFIRST* e *WM_KEYLAST* como valores de filtro para recuperar todas as mensagens de teclado; você pode usar as constantes *WM_MOUSEFIRST* e *WM_MOUSELAST* para recuperar todas as mensagens do mouse.

A função *PeekMessage* normalmente não remove *WM_PAINT* da fila. As mensagens *WM_PAINT* permanecem na fila até que o programa as processe. No entanto, se uma mensagem *WM_PAINT* tiver uma região de atualização, *PeekMessage* a removerá da fila.

COMPREENDENDO A FUNÇÃO POSTMESSAGE 1292

Em dicas anteriores, você aprendeu vários modos de seus programas poderem recuperar mensagens da fila de mensagem. Algumas vezes, você pode querer colocar mensagens dentro da fila de mensagens do programa. A função *PostMessage* coloca (envia) uma mensagem na fila de mensagem associada com o encadeamento que criou a janela especificada e depois retorna sem esperar que o encadeamento processe a mensagem. As chamadas à função *GetMessage* ou *PeekMessage* recuperam mensagens da fila de mensagem que você encaminhou anteriormente com *PostMessage*. Você usará a função *PostMessage* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL PostMessage (
   HWND hJan,      // indicativo da janela de destino
   UINT Msg,       // mensagem a encaminhar
   WPARAM wParam,  // primeiro parâmetro da mensagem
   LPARAM lParam   // segundo parâmetro da mensagem
);
```

A função *PostMessage* aceita os parâmetros detalhados na Tabela 1292 1.

Tabela 1292.1 Os parâmetros para a função ***PostMessage***.

Parâmetro	Descrição
hJan	Identifica a janela cujo procedimento de janela deve receber a mensagem.
Msg	Especifica a mensagem para *PostMessage* colocar dentro da fila.
wParam	Especifica informações adicionais específicas para a mensagem.
lParam	Especifica informações adicionais específicas para a mensagem.

Dois valores para o parâmetro *hJan* têm significado especial para a função *PostMessage*, como detalha a Tabela 1292.2.

Tabela 1292.2 Os valores especiais para a função ***PostMessage***.

Valor	Significado
HWND_BROADCAST	A função *PostMessage* envia a mensagem para todas as janelas de alto nível no sistema, incluindo janelas desabilitadas ou invisíveis sem proprietário, janelas sobrepostas e janelas instantâneas. A função *PostMessage* não envia a mensagem para as janelas-filha.
NULL	A função se comporta como uma chamada à *PostThreadMessage* com o parâmetro *dwThreadId* definido com o identificador do encadeamento atual.

Os aplicativos que precisam usar *HWND_BROADCAST* para comunicar devem usar a função *RegisterWindowMessage* para obter uma mensagem única para a comunicação interaplicativa.

Se você enviar uma mensagem no intervalo abaixo de *WM_USER* para as funções de mensagem assíncrona (*PostMessage*, *SendNotifyMessage* e *SendMessageCallback*), certifique-se de que os parâmetros da mensagem não incluem ponteiros. Caso contrário, as funções retornarão antes que o encadeamento receptor tenha tido uma chance de processar a mensagem, e o transmissor liberará a memória antes que o programa a use.

1293 COMPREENDENDO A FUNÇÃO SENDMESSAGE

A dica anterior mostrou como usar a função *PostMessage* para colocar uma mensagem dentro da fila de mensagem do encadeamento. Como você aprendeu, *PostMessage* retorna o processamento para o programa chamador sem aguardar por uma resposta a partir do encadeamento que recebe a mensagem. Algumas vezes, seus programas podem requerer uma resposta do encadeamento receptor antes que o processamento possa continuar dentro do encadeamento chamador. Você pode usar a função *SendMessage* para controlar quando o processamento do aplicativo retorna para o aplicativo chamador. A função *SendMessage* envia uma mensagem que você especificar para uma janela ou janelas. A função *SendMessage* chama o procedimento de janela para a janela especificada, e não retorna o processamento para o programa chamador até que o procedimento de janela processe a mensagem. A função *PostMessage*, em contraste, encaminha uma mensagem para a fila de mensagem de um encadeamento e retorna imediatamente. Seus programas usarão a função *SendMessage* como mostrado no protótipo a seguir:

```
LRESULT  SendMessage(
   HWND   hJan,     // indicativo da janela de destino
   UINT   Msg,      // mensagem a enviar
   WPARAM wParam,   // primeiro parâmetro da mensagem
   LPARAM lParam    // segundo parâmetro de mensagem
);
```

Os parâmetros para a função *SendMessage* são idênticos aos parâmetros para a função *PostMessage*. No entanto, se você definir o parâmetro *hJan* para *HWND_BROADCAST*, seu programa enviará a mensagem para todas as janelas de alto nível no sistema, incluindo as janelas desabilitadas, as invisíveis sem proprietário, as janelas sobrepostas e as janelas de ativação instantânea. No entanto, seu programa não enviará a mensagem para as janelas-filha.

Os aplicativos que precisam usar *HWND_BROADCAST* para comunicar devem usar a função *RegisterWindowMessage* para obter uma mensagem única para a comunicação interaplicativa.

Se o encadeamento chamador criou a janela especificada, o Windows chama o procedimento de janela imediatamente, como uma sub-rotina. Se um encadeamento diferente criou a janela especificada, o Windows alterna para esse encadeamento e chama o procedimento de janela apropriado. O Windows processa as mensagens enviadas entre os encadeamentos somente quando o encadeamento receptor executa o código de recuperação de mensagem. O Windows bloqueia o encadeamento transmissor até que o receptor processe a mensagem.

1294 USANDO A FUNÇÃO REPLYMESSAGE

Em dicas anteriores você aprendeu como usar as funções *PostMessage* e *SendMessage* dentro de seus programas. Seus programas devem usar a função *ReplyMessage* para responder a uma mensagem enviada por meio da mensagem *SendMessage* sem retornar o controle para a função que chamou *SendMessage*, como mostrado aqui:

```
BOOL ReplyMessage (
   LRESULT lResult  // resposta específica de mensagem
);
```

Chamando a função *ReplyMessage*, o procedimento de janela que recebe a mensagem permite que o encadeamento que chamou *SendMessage* continue a rodar como se o encadeamento que está recebendo a mensagem tivesse retornado o controle. O encadeamento que chama a função *ReplyMessage* também continua a executar.

Se o programa não enviou a mensagem por meio de *SendMessage*, ou se o mesmo encadeamento enviou a mensagem, *ReplyMessage* não tem efeito.

GANCHOS DE MENSAGENS 1295

As mensagens estão no centro de como o sistema operacional Windows gerencia os programas. Como também aprendeu, você pode usar diferentes funções da API para passar mensagens de um encadeamento para outro. No entanto, algumas vezes, você pode querer que um determinado programa intercepte todas as mensagens que o Windows envia para outro programa. Os *ganchos de mensagem* lhe permitem interceptar as mensagens que vão para um programa dentro de um programa diferente. O programa a seguir pode atuar, modificar ou até parar as mensagens que intercepta. Você freqüentemente interceptará mensagens dentro de uma biblioteca da ligações de dados (DLL), um tipo especial de arquivo de suporte, para fornecer serviços adicionais de processamento de mensagem sem reduzir a velocidade do programa principal. O escopo de um gancho depende do tipo dele. Você possa definir alguns ganchos somente com escopo no sistema, embora outros você pode definir somente para um encadeamento específico, como mostrado na lista a seguir. A Tabela 1295 lista os tipos de ganchos na ordem em que o sistema operacional pega os ganchos.

Tabela 1295 Escopo dos ganchos do sistema.

Gancho	Escopo
WH_CALLWNDPROC	Encadeamento ou sistema
WH_CALLWNDPROCRET	Encadeamento ou sistema
WM_CBT	Encadeamento ou sistema
WH_DEBUG	Encadeamento ou sistema
WH_GETMESSAGE	Encadeamento ou sistema
WH_JOURNALPLAYBACK	Somente sistema
WM_JOURNALRECORD	Somente sistema
WH_KEYBOARD	Encadeamento ou sistema
WH_MOUSE	Encadeamento ou sistema
WH_MSGFILTER	Encadeamento ou sistema
WH_SHELL	Encadeamento ou sistema
WH_SYSMSGFILTER	Somente sistema

Para um tipo de gancho especificado, o Windows chama os ganchos dos encadeamentos primeiro, depois os ganchos do sistema.

USANDO A FUNÇÃO SETWINDOWSHOOKEX 1296

Na dica anterior, você aprendeu sobre os ganchos de mensagem, que seus programas podem usar para interceptar os ganchos voltados para outros aplicativos. A função *SetWindowsHookEx* instala um *procedimento de gancho* definido pelo aplicativo em uma *cadeia de gancho*. Seus programas podem usar um procedimento de gancho para monitorar o sistema para certos tipos de eventos. Quando você instala um procedimento de gancho, o Windows associa os eventos que esse procedimento de gancho monitora com um encadeamento específico ou com todos os encadeamentos no sistema. Você usará a função *SetWindowsHookEx* dentro de seus programas, como mostrado no protótipo a seguir:

```
HHOOK SetWindowsHookEx(
   int idHook,       // tipo de gancho a instalar
   HOOKPROC lpfn,    // endereço do proced. de gancho
   HINSTANCE hMod,   // indicativo da ocorrência
   DWORD dwThreadId  // identidade do encadeamento para o qual instalar o
                     // gancho
);
```

O parâmetro *idHook* especifica o tipo de procedimento de gancho para o programa instalar. O parâmetro pode ser um dos valores listados na Tabela 1296. O parâmetro *lpfn* aponta para o procedimento de gancho. Se o parâmetro *dwThreadId* for 0 ou especificar o identificador de um encadeamento que um processo diferente

criou, o parâmetro *lpfn* precisará apontar para um procedimento de gancho em uma biblioteca de ligações dinâmicas (DLL). Caso contrário, *lpfn* poderá apontar para um procedimento de gancho no código associado com o processo atual. O parâmetro *hMod* identifica a biblioteca de ligações dinâmicas que contém o procedimento de gancho para o qual o parâmetro *lpfn* aponta. Você precisará definir o parâmetro *hMod* como *NULL* se o parâmetro *dwThreadId* especificar um encadeamento que o processo atual criou e se o procedimento de gancho estiver dentro do código associado com o processo atual. O parâmetro *dwThreadId* especifica o identificador do encadeamento com o qual seu programa associará o procedimento de gancho. Se o parâmetro for 0, seu programa associará o procedimento de gancho com todos os encadeamentos existentes.

Como você poderia esperar, pode definir uma variedade de diferentes tipos de ganchos. Você determinará qual tipo de gancho o sistema definirá com o valor que você coloca dentro do parâmetro *idHook*. A Tabela 1296 detalha os possíveis valores para o parâmetro *idHook*.

Tabela 1296 Os valores possíveis para o parâmetro ***idHook***.

Valores	Descrição
WH_CALLWNDPROC	Instala um procedimento de gancho que monitora as mensagens antes que o sistema as envie para o procedimento de janela de destino.
WH_CALLWNDPROCRET	Instala um procedimento de gancho que monitora as mensagens após o procedimento de janela de destino processá-las.
WH_CBT	Instala um procedimento de gancho que recebe notificações úteis para um aplicativo de treinamento baseado em computador (CBT).
WH_DEBUG	Instala um procedimento de gancho útil para depurar outros procedimentos de gancho.
WH_GETMESSAGE	Instala um procedimento de gancho que monitora mensagens encaminhadas a uma fila de mensagem.
WH_JOURNALPLAYBACK	Instala um procedimento de gancho que encaminha mensagens que um procedimento de gancho *WH_JOURNALRECORD* gravou anteriormente.
WH_JOURNALRECORD	Instala um procedimento de gancho que registra as mensagens de entrada encaminhadas para a fila de mensagens do sistema. Esse gancho é útil para gravar macros.
WH_KEYBOARD	Instala um procedimento de gancho que monitora as mensagens de digitação
WH_MOUSE	Instala um procedimento de gancho que monitora as mensagens do mouse.
WH_MSGFILTER	Instala um procedimento de gancho que monitora as mensagens geradas como resultado de um evento de entrada em uma caixa de diálogo, caixa de mensagem ou barra de rolagem.
WH_SHELL	Instala um procedimento de gancho que recebe notificações úteis para os aplicativos de interface.
WH_SYSMSGFILTER	Instala um procedimento de ganhco que monitora mensagens geradas como resultado de um evento de entrada em uma caixa de diálogo, caixa de mensagem, menu, ou barra de rolagem. O procedimento de gancho monitora essas mensagens para todos os aplicativos no sistema.

Um erro poderá ocorrer se o parâmetro *hMod* for *NULL* e o parâmetro *dwThreadId* for 0 ou especificar o identificador de um encadeamento que outro processo criou.

Chamar a função *CallNextHookEx* para encadear para o próximo procedimento de gancho é opcional. No entanto, como você aprenderá, se você não usar a função *CallNextHookEx*, outros aplicativos que instalaram anteriormente ganchos não receberão as notificações de gancho e, como resultado, poderão se comportar incorretamente. Você deve chamar *CallNextHookEx* a não ser que absolutamente precise impedir que outros aplicativos vejam uma notificação.

Antes de terminar, um aplicativo precisa chamar a função *UnhookWindowsHookEx* para liberar recursos do sistema associados com o gancho.

Os ganchos do sistema são um recurso compartilhado, e instalar um afeta todos os aplicativos. Todas as funções de gancho do sistema precisam estar em bibliotecas. Você deve restringir os ganchos do sistema a aplicativos de propósito especial ou usá-los como uma ajuda no desenvolvimento durante a depuração do aplicativo. As bibliotecas que não precisam mais de um gancho devem remover o procedimento de gancho.

COMPREENDENDO A FUNÇÃO EXITWINDOWSEX 1297

Em dicas anteriores, você aprendeu sobre o uso de mensagens dentro de seus programas. Você encontrará um modo ligeiramente diferente de usar mensagens dentro da função *ExitWindowsEx*. A função *ExitWindowsEx* desliga ou reinicia o sistema. Seus programas usarão a função *ExitWindowsEx* como mostrado no protótipo a seguir:

```
BOOL ExitWindowsEx(
   UINT  uFlags,          // operação de desligar
   DWORD dwReserved;      // reservado
);
```

Dentro da função *ExitWindowsEx*, o parâmetro *uFlags* especifica o tipo de desligamento. O parâmetro *uFlags* precisa ser alguma combinação dos valores que a Tabela 1297 detalha. O Windows atualmente ignora o parâmetro *dwReserved* e o reserva para futuros desenvolvimentos. A Tabela 1297 lista os valores possíveis para o parâmetro *uFlags*.

Tabela 1297 Os valores possíveis para o parâmetro ***uFlags***.

Sinalizador	Descrição
EWX_FORCE	Força os processos a terminar. Quando o programa define esse sinalizador, o Windows não envia as mensagens *WM_QUERYENDSESSION* e *WM_ENDSESSION* para os aplicativos que estiverem em execução no sistema no momento. Isso pode fazer com que os aplicativos percam dados. Portanto, você somente deverá usar esse sinalizador em uma emergência.
EWX_LOGOFF	Desliga todos os processos que estiverem rodando no contexto de segurança do processo que chamou a função *ExitWindowsEx*. Em seguida, desliga o usuário.
EWX_POWEROFF	Desliga o sistema e a energia. O sistema precisa suportar o recurso do corte da energia.
Nota: *No Windows NT, o processo chamador precisa ter o privilégio SE_SHUTDOWN_NAME. O Windows 95 não suporta ou requer privilégios de segurança.*	
EWX_REBOOT	Desliga e depois reinicia o sistema.
Nota: *No Windows NT, o processo chamador precisa ter o privilégio SE_SHUTDOWN_NAME.*	
EWX_SHUTDOWN	Desliga o sistema para um ponto em que é seguro desligar a energia. O sistema esvazia todos os buffers de arquivo para o disco e pára todos os processos que estiverem em execução.

A função *ExitWindowEx* retorna tão logo tiver iniciado o desligamento. O desligamento então procede assincronamente (em outras palavras, seus programas podem efetuar processamento adicional durante o processo de desligamento). Durante uma operação de desligamento, o sistema dá aos aplicativos que ele desliga uma quantidade específica de tempo para responder à solicitação do desligamento. Se o tempo expirar, o Windows exibirá uma caixa de diálogo que permite que o usuário force o desligamento do aplicativo, repita o desligamento ou cancele a solicitação de desligamento. Se você especificar o valor *EWX_FORCE*, o Windows sempre forçará os aplicativos a fechar, e não exibirá a caixa de diálogo para o usuário.

A função *ExitWindowsEx* envia uma mensagem de notificação separada — *CTRL_SHUTDOWN_EVENT* ou *CTRL_LOGOFF_EVENT*, conforme a situação justificar — para os *processos de console*. Um processo de console roteia essas mensagens para suas funções *HandlerRoutine*, que chama a função *SetConsoleCtrlHandler* para acrescentar e remover. *ExitWindowsEx* envia essas mensagens de notificação assincronamente; portanto, um apli-

cativo não poderá assumir que as funções *HandlerRoutine* trataram as mensagens de notificação de console quando uma chamada à *ExitWindowsEx* retornar.

Nota: Para desligar ou reiniciar um sistema Windows NT, o processo chamador precisa usar a função AdjustTokenPrivileges para habilitar o privilégio SE_SHUTDOWN_NAME. Os processos de console rodam somente em um servidor Windows NT. O Windows 95 não suporta ou requer os privilégios de segurança.

1298 COMPREENDENDO OS TIPOS DE MENU

Como você viu em dicas anteriores, um componente principal da arquitetura do Windows 95 e do Windows NT é a *barra de menu*. A barra de menu é, basicamente, um repositório para um recurso de menu. Como você viu, os recursos de menu permitem que seus programas recebam entrada textual e orientada pelo mouse. Tipicamente, você agrupará menus sob categorias específicas e fornecerá ao usuário múltiplas opções dentro dessas categorias.

Os programas Windows usam dois tipos básicos de menus: *menus de alto nível* e *menus instantâneos*. O menu de alto nível (também chamado menu "principal" do programa) é um conjunto de comandos visíveis na barra de menu de uma janela todo o tempo, assumindo que o programa use um menu. Para os programas mais complexos, pode não ser sempre possível ou prático colocar todas as opções de menu que o programa pode requerer na barra de menu. Nesses casos, seus programas podem usar menus de ativação instantânea (também chamados "submenus" ou "menus suspensos"). Quando você selecionar uma opção a partir da barra de menu principal que gera um menu instantâneo, o Windows também estenderá automaticamente o menu instantâneo. A Figura 1298 mostra um menu e um submenu simples de alto nível.

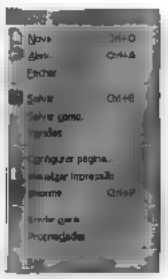

Figura 1298 Um menu de alto nível e um submenu simples.

Nota: Não confunda menus de ativação instantânea no contexto da API do Windows com menus sensíveis ao contexto, que muitos livros também referenciam como menus de ativação instantânea. Um menu sensível ao contexto é um tipo especial de menu, sobre o qual você aprenderá mais em dicas posteriores.

1299 COMPREENDENDO A ESTRUTURA DE UM MENU

Como você aprendeu, o Windows compõe um menu de programa de uma ou mais seleções de menu de alto nível, uma ou mais seleções sob cada seleção de menu de alto nível (chamadas *opções* neste livro) e uma ou mais seleções sob cada uma das seleções (chamadas *subopções*). Por exemplo, a Figura 1299 mostra uma árvore de estrutura de exemplo para um conjunto de menus para um programa simples. Observe como, na hierarquia de menu, os menus de alto nível têm um número variável de opções e que somente algumas das opções têm outras subopções, como mostrado aqui:

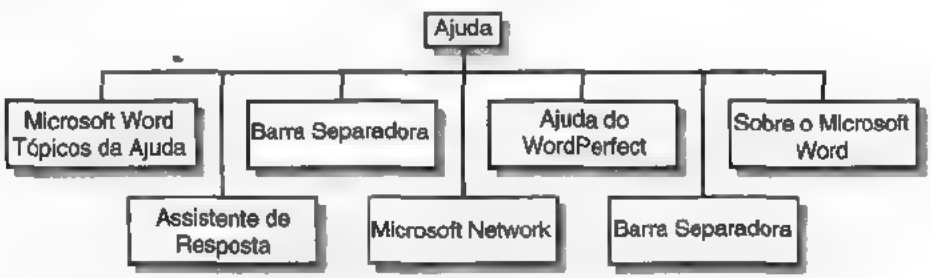

Figura 1299 A estrutura hierárquica dos menus do programa.

Adicionalmente, é importante observar que, em uma boa estrutura de menu, há apenas um modo de alcançar qualquer dada opção ou subopção. Manter o menu estritamente hierárquico protege o usuário da confusão entre subopções em múltiplos menus. Quando você usa uma estrutura de menu estritamente hierárquica, o menu do seu programa deve parecer estruturalmente idêntico ao diagrama na Figura 1299.1, como mostrado na Figura 1299.2.

Figura 1299.2 O menu do programa real.

CRIANDO UM MENU DENTRO DE UM ARQUIVO DE RECURSO 1300

Como aprendeu em todo este livro, você mais frequentemente criará seus menus dentro de um arquivo de recurso. Como várias dicas anteriores mencionaram, as ferramentas de desenvolvimento para C++ mais modernas lhe permitirão criar menus usando métodos mais simples de arrastar-e-colar. No entanto, é importante compreender como seus programas criam menus caso você precise criar um manualmente. A listagem a seguir, por exemplo, mostra a construção de menu típica dentro de um arquivo de recursos:

```
MEUAPLIC MENU DISCARDABLE
BEGIN
    POPUP "&Arquivo"
    BEGIN
        MENUITEM "&Sair",    IDM_SAIR
    END
    POPUP "&Testar!"
    BEGIN
       MENUITEM "Item &1", IDM_ITEM1
       MENUITEM "Item &2", IDM_ITEM2
       MENUITEM "Item &3", IDM_ITEM3
    END
    POPUP "A&juda"
    BEGIN
        MENUITEM "&Sobre Meu Aplicativo...",  IDM_SOBRE
    END
END
```

Nas dicas a seguir, você examinará em detalhes a definição de menu do arquivo de recursos, e também como usará a definição dentro de seus programas. Especificamente, na dica a seguir, você examinará como seus arquivos de recursos usam os descritores *POPUP* e *MENUITEM*. No entanto, é importante observar o descritor *DISCARDABLE* dentro da declaração de recursos anterior. O descritor *DISCARDABLE* informa o linkeditor do compilador que ele deve descartar as informações de recursos iniciais sobre o menu após o programa registrar o menu dentro da classe da janela. Você quase sempre usará o descritor *DISCARDABLE* com suas definições de menu para conservar memória e melhorar a velocidade de processamento do programa.

COMPREENDENDO OS DESCRITORES POPUP E MENUITEM 1301

Como visto na dica anterior, a seção de descrição de menu do arquivo de recursos inicia com o comando *BEGIN* e termina com o comando *END*. Dentro de uma descrição de menu, você pode ter qualquer número de itens *POPUP* e *MENUITEM*. No entanto, deve ter cuidado para não criar itens de alto nível demais, pois eles podem não caber na janela quando esta estiver em seu tamanho normal.

Dentro da descrição do menu, o descritor *POPUP* indica o nível superior do menu. Por exemplo, o item *Testar!* está no topo de um menu no programa na dica anterior. O uso de um & dentro da string indica que o usuário pode usar uma *tecla aceleradora*, em vez do mouse, para acessar o item de menu. Os programadores frequentemente referenciam qualquer tecla aceleradora como um "atalho de teclado". No caso de um menu *Testar!*, a tecla aceleradora é Alt+K.

O arquivo de recursos usa o descritor *MENUITEM* para listar cada item dentro do menu. Por exemplo, a descrição do menu *Testar!* que a dica anterior usou contém três itens de menu. Item #1, Item #2 e Item #3. Observe que cada item de menu inclui uma referência constante. No caso do Item #1, a referência constante é *IDM_ITEM1*. Dentro de sua rotina *ProcJan*, seu programa primeiro pegará a mensagem *WM_COMMAND*. Após seu programa pegar a mensagem *WM_COMMAND*, a palavra de baixa ordem do valor *wParam* que *ProcJan* recebe conterá a constante identificadora para o item de menu selecionado pelo usuário. Por exemplo, para processar a seleção da opção *IDM_ITEM1*, seu programa usará dois comandos *switch*. O primeiro comando verificará que a mensagem é do tipo *WM_COMMAND*. O segundo comando verificará o valor dentro de *wParam* para checar se o item de menu que o usuário selecionou foi a opção Item #1, como mostrado aqui:

```
switch(uMsg)
   {
      case WM_COMMAND :
           switch(LOWORD(wParam))
           {
              case ITEM1 :
                           // algum processamento
              case ITEM2 :
                           // mais processamento
```

1302 ACRESCENTANDO UM MENU À JANELA DE UM APLICATIVO

Como você aprendeu, definir um menu dentro do arquivo de recursos (.RC) não torna automaticamente o menu usável ou anexa o menu à janela do aplicativo. Normalmente, você anexará o menu do aplicativo à definição da classe da janela dentro da função *WinMain* antes de chamar a função *RegisterClass*, como você viu em dicas anteriores. Lembre-se você define o elemento *lpszMenuName* das estruturas *WNDCLASS* e *WNDCLASSEX* para efetuar a anexação de amarração precoce. A função *RegisterClass* então associa o nome de menu com qualquer janela que o programa depois criar com essa classe.

Você também pode usar as funções *CreateWindow* e *LoadMenu* para anexar o menu a uma janela. Quando você chamar a função *CreateWindow*, poderá definir o parâmetro *hMenu* com o valor que a função *LoadMenu* retornará. Depois, você chamará *LoadMenu* com o nome de menu do arquivo de recurso como o parâmetro da função, como mostrado aqui:

```
if (LOWORD(wParam) == IDM_NEW )
   hNovoMenu = LoadMenu(hCop, "NOVOMENU" );
else
   hNovoMenu = LoadMenu(hCop, "MENUANTIGO" );
```

Finalmente, você poderá usar as funções SetMenu e LoadMenu para anexar um menu a uma janela após ter criado a janela. Novamente, você invocará a função *SetMenu* com o parâmetro de indicativo *hMenu*. *SetMenu* associa o menu ao qual o indicativo *hMenu* aponta com a janela do aplicativo que está atualmente aberta, como mostrado aqui:

```
SetMenu(hJan, hNovoMenu );
```

1303 ALTERANDO OS MENUS DENTRO DE UM APLICATIVO

Em dicas anteriores, você aprendeu como seus programas podem usar uma variedade de métodos para anexar menus a um aplicativo. No entanto, ao longo do projeto de seu programa Windows, você poderá desenvolver aplicativos mais complexos que comumente requerem alterações no menu enquanto o aplicativo estiver em execução. A API Win32 *fornece um conjunto amplo de funções que seus programas podem usar para alterar um* menu durante a execução de um aplicativo. Essencialmente, as funções lhe permitem modificar todo aspecto do menu após você ter anexado o menu à janela.

As alterações mais comuns que seus programas farão nos menus serão alterar a string que o item de menu exibe, colocar e remover marcas de verificação ao lado dos itens de menu, habilitar e desabilitar itens de menu e excluir e acrescentar itens no menu. A função *ModifyMenu* lhe permite efetuar diversas dessas operações a partir de dentro de uma única chamada de função. Alternativamente, você pode usar funções mais específicas, tais como *DeleteMenu* ou *CheckMenuItem* para modificar seus itens de menu. O CD-ROM que acompanha este li-

vro inclui o programa *Remove.cpp*, que usa comandos de menu para permitir que o usuário acrescente e exclua itens de um menu.

COMPREENDENDO AS MENSAGENS GERADAS POR MENUS 1304

Cada vez que o usuário selecionar um item no menu dentro de seus programas, o Windows enviará a mensagem *WM_COMMAND* para o laço de mensagem do programa. Dentro do laço de mensagem, seu programa precisa então verificar a palavra menos significativa do valor *DWORD wParam* para determinar qual item de menu o usuário selecionou.

Como regra, *WM_COMMAND* é a única mensagem que o Windows enviará para seus programas como resultado de uma seleção de menu do usuário. Se o usuário selecionar um item de menu do sistema, por exemplo, o Windows, em vez disso, enviará a mensagem *WM_SYSCOMMAND* (que você então provavelmente tratará dentro do procedimento *DefWindowProc*). Seu aplicativo pode, por outro lado, requerer que seu laço de mensagem processe as mensagens *WM_INITMENU* e *WM_INITMENUPOPUP* — ambas as quais o sistema envia para a janela imediatamente antes de o sistema ativar um menu (ou um menu Principal ou Popup, dependendo da mensagem). Capturar as mensagens *WM_INITMENU* e *WM_INITMENUPOPUP* permitirá que seu aplicativo modifique um menu ou menus, se quaisquer alterações forem necessárias, imediatamente antes de o Windows exibir o menu do aplicativo.

O menu também enviará a mensagem *WM_MENUSELECT* cada vez que o usuário selecionar um item de menu. A mensagem *WM_MENUSELECT* é mais poderosa e versátil que a mensagem *WM_COMMAND* porque o Windows a gera mesmo se o item de menu estiver atualmente desabilitado. No entanto, você normalmente usará apenas a mensagem *WM_MENUSELECT* para exibir ajuda de menu sensível ao contexto.

COMPREENDENDO A FUNÇÃO LOADMENU 1305

Como você aprendeu na Dica 1302, seus programas podem usar a função *LoadMenu* para carregar um menu que você definiu anteriormente dentro de um arquivo de recursos. Tipicamente, você seguirá a chamada de *LoadMenu* com uma chamada à *SetMenu* ou usará a chamada *LoadMenu* dentro de uma chamada à *CreateWindow*. A função *LoadMenu* carrega o recurso de menu que o parâmetro *lpNomeMenu* especifica a partir do arquivo executável (.EXE) que o Windows associa com a ocorrência do aplicativo. Você usará a função *LoadMenu* dentro de seus programas, como mostrado no protótipo a seguir:

```
HMENU LoadMenu (
   HINSTANCE hCopia,    // indicativo de ocorrência de aplicativo
   LPCTSTR  lpNomeMenu // string de nome de menu ou identificador de
                        recurso de menu
```

LoadMenu retorna um indicativo *HMENU* e aceita como seu parâmetro o indicativo para a ocorrência do módulo que contém o recurso de menu que *LoadMenu* deverá carregar e um ponteiro para uma string terminada por *NULL*, que contém o nome do recurso de menu. Em vez de usar um ponteiro string para o nome do menu, você pode usar uma *DWORD* como o segundo parâmetro (o ponteiro para o nome do menu). Em tal caso, a *DWORD* pode consistir de um identificador de recurso, com o identificador real na *WORD* menos significativa e zero na *WORD* mais significativa. Para criar o valor *DWORD*, em lugar da constante ponteiro string, use a macro *MAKEINTRESOURCE*.

Para compreender melhor o processamento que *LoadMenu* executa, considere o fragmento de código a seguir do programa *2_menus*, que se alterna entre dois menus, com base na seleção do usuário. Tanto o arquivo de recursos quanto o arquivo de programas diferem de seus arquivos *generico* correspondentes. O arquivo *2_menus.rc* inclui as seguintes definições de menu:

```
MENUANTIGO MENU DISCARDABLE
BEGIN
    POPUP "&Arquivo"
    BEGIN
        MENUITEM "&Sair",     IDM_SAIR
    END
    MENUITEM "&Novo Menu!"    IDM_NOVO

MENUNOVO MENU DISCARDABLE
```

```
BEGIN
    POPUP "&Arquivo"
    BEGIN
        MENUITEM "&Sair",       IDM_SAIR
    END
    MENUITEM "&Menu Antigo!",  IDM_ANTIGO
END
```

A primeira declaração cria o menu *MENUANTIGO* com a opção *Novo Menu*. Quando o usuário selecionar essa opção, o programa então mudará o menu dentro da janela do programa do menu *MENUANTIGO* para o menu *MENUNOVO*. Como você pode esperar, o processamento que modifica o menu do aplicativo de *MENUANTIGO* para *MENUNOVO* está dentro da função *ProcJan* do arquivo *2_menus.cpp*.

Como você pode ver, o programa verifica o valor constante que a palavra menos significativa do valor *wParam* contém. Dependendo do resultado, o programa carrega o outro menu (em outras palavras, se o resultado indicar que o programa está atualmente exibindo o recurso *MENUANTIGO*, o programa exibirá o recurso *MENUNOVO* e vice-versa). O programa então usa a função *SetMenu* para se alterar entre as funções. Finalmente, o programa chama a função *DrawMenuBar* após completar a alternação, o que garante que o Windows redesenhe o menu recém-carregado.

Nota: Seus programas precisam usar a função DestroyMenu antes que o aplicativo seja fechado para destruir o menu e liberar a memória livre que o menu carregado ocupava.

1306 USANDO A FUNÇÃO MODIFYMENU

Como você aprendeu na Dica 1303, seus programas podem usar a função *ModifyMenu* para fazer uma variedade de modificações nos seus menus após seus programas os anexarem a uma janela. A função *ModifyMenu* muda um item de menu existente. Seus programas podem usar a função *ModifyMenu* para especificar o conteúdo, a aparência e o comportamento de cada item dentro de um menu. Você usará a função *ModifyMenu* dentro de seus programas como mostrado no protótipo a seguir:

```
BOOL ModifyMenu(
    HMENU hMnu,         // indicativo do menu
    UINT uPosicao,      // item de menu a modificar
    UINT uFlags,        // sinalizadores de item de menu
    UINT uIDNovoItem,   // identificador de item de menu ou
                        // indicativo de menu suspenso
    LPCTSTR lpNovoItem  // conteúdo do item de menu
);
```

A função *ModifyMenu* aceita os parâmetros detalhados na Tabela 1306.1

Tabela 1306.1 Os parâmetros que a função *ModifyMenu* aceita.

Parâmetros	Descrição
hMnu	Identifica o menu que você quer modificar.
uPosicao	Especifica o item de menu que você quer modificar de acordo com os valores do parâmetro *uFlags*.
uFlags	Especifica os sinalizadores que controlam a interpretação do parâmetro *uPosicao* e o conteúdo, a aparência e o comportamento do item do menu. Esse parâmetro precisa ser uma combinação de um dos valores requeridos que a Tabela 1306.2 detalha e pelo menos um dos valores da Tabela 1306.3.
uIDNewItem	Especifica se o identificador do item de menu modificado, ou se o parâmetro *uFlags* tem um sinalizador *MF_POPUP* ligado, o indicativo do menu ou o submenu suspenso.
lpNewItem	Aponta para o conteúdo do item de menu modificado. A interpretação deste parâmetro depende de o parâmetro *uFlags* incluir o sinalizador *MF_BITMAP*, *MF_OWNERDRAW*, ou *MF_STRING*.

Como detalha a Tabela 1306.1, você precisa passar um valor dentro do parâmetro *uFlags* necessário. O valor *uFlags* precisa ser uma combinação dos valores listados na Tabela 1306.2 e um ou mais dos valores listados na Tabela 1306.3.

Tabela 1306.2 Os valores requeridos possíveis para o parâmetro ***uFlags***.

Valor	Significado
MF_BYCOMMAND	Indica que o parâmetro *uPosicao* dá o identificador do item de menu. O sinalizador *MF_BYCOMMAND* é o padrão se você não especifica o sinalizador *MF_BYCOMMAND* nem *MF_BYPOSITION*.
MF_BYPOSITION	Indica que o parâmetro *uPosicao* dá a posição relativa com base em zero do item de menu.

Além do valor requerido para o parâmetro *uFlags*, você também precisa usar o operador *OU* bit a bit ou atribuir um ou mais dos valores que a Tabela 1306.3 detalha para o item de menu.

Tabela 1306.3 Os valores possíveis para o parâmetro ***uFlags***.

Valor	Significado
MF_BITMAP	Contém o indicativo do mapa de bits.
MF_OWNERDRAW	Contém um valor de 32 bits fornecido por um aplicativo que o Windows usa para manter dados adicionais relacionados com o item de menu. O valor está no membro *itemDados* da estrutura apontada pelo parâmetro *lparam* da mensagem *WM_MEASUREITEM* ou *WM_DRAWITEM* que o programa envia quando você cria o item de menu ou atualiza a aparência dele
MF_CHECKED	Coloca uma marca de verificação ao lado do item. Se seu aplicativo fornece bitmaps de marca de verificação (veja a função *SetMenuItemBitmaps*), esse sinalizador exibe um mapa de bits ao lado do item de menu.
MF_DISABLED	Desabilita o item de menu para que o usuário não possa selecioná-lo, mas esse sinalizador não torna o item de menu cinza
MF_ENABLED	Habilita o item de menu e restaura o de seu estado acinzentado para que o usuário possa selecioná-lo
MF_GRAYED	Desabilita o item de menu e torna-o cinza para que o usuário não possa selecioná-lo.
MF_MENUBARBREAK	Funciona da mesma forma que o sinalizador *MF_MENUBREAK* para uma barra de menu. Para um menu suspenso, submenu, ou menu de atalho, uma linha vertical separa as duas colunas da coluna antiga.
MF_MENUBREAK	Coloca o item em uma nova linha (para barras de menu) ou em uma nova coluna (para um menu suspenso, submenu, ou menu de atalho) sem separar as colunas.
MF_OWNERDRAW	Especifica que o item é um item desenhado pelo usuário. Antes que o aplicativo exiba o menu pela primeira vez, a janela que possui o menu recebe uma mensagem *WM_MEASUREITEM* para recuperar a largura e a altura do item de menu. O programa então enviará uma mensagem *WM_DRAWITEM* para o procedimento de janela da janela proprietária sempre que seu programa precisar atualizar a aparência do item de menu.
MF_POPUP	Especifica que o item de menu abre um menu suspenso ou um submenu. O parâmetro *uIDNewItem* especifica o indicativo do menu ou submenu suspenso. Você usará esse sinalizador para acrescentar um nome de menu a uma barra de menu ou um item de menu que abre um submenu para um menu suspenso, um submenu ou um menu de atalho

Tabela 1306.3 Os valores possíveis para o parâmetro *uFlags*. (*Continuação*)

Valor	Significado
MF_SEPARATOR	Desenha uma linha divisória horizontal. Você usará esse sinalizador somente ao acrescentar itens a um menu suspenso, um submenu ou menu de atalho. Seus programas não podem acinzentar, desabilitar ou destacar a linha O Windows ignora os parâmetros *lpNewItem* e *uIDNewItem*.
MF_UNCHECKED	Não coloca uma marca de verificação ao lado do item (o padrão). Se seu aplicativo fornece mapas de bit de marca de verificação (veja a função *SetMenuItemBitmap*), esse sinalizador exibe um mapa de bits não-assinalado ao lado do item de menu
MF_STRING	Contém um ponteiro para uma string terminada por *NULL* (o padrão).

Se *ModifyMenu* substituir um item de menu que abre um menu suspenso ou um submenu, a função destrói o menu ou o submenu antigo e libera a memória que o menu antigo usava. Adicionalmente, o aplicativo precisará chamar a função *DrawMenuBar* sempre que um menu mudar, independentemente se o menu estiver ou não exibido em uma janela. Para modificar os atributos de itens de menu existentes, é muito mais rápido usar as funções *CheckMenuItem* e *EnableMenuItem*.

Nota: *A API do Windows não lhe permitirá usar os seguintes grupos de sinalizadores juntos quando você chamar ModifyMenu.*

- *MF_BYCOMMAND* e *MF_BYPOSITION*
- *MF_DISABLED, MF_ENABLED* e *MF_GRAYED*
- *MF_BITMAP, MF_STRING, MF_OWNERDRAW* e *MF_SEPARATOR*
- *MF_MENUBARBREAK* e *MF_MENUBREAK*
- *MF_CHECKED* e *MF_UNCHECKED*

Para compreender melhor o processamento que a função *ModifyMenu* executa, considere o programa *Mod_Menu cpp*, contido no CD-ROM que acompanha este livro. O programa *Mod_Menu.cpp* altera o item *Testar!* do item de menu *Novo Item!* quando o usuário o seleciona, depois processa o item de menu *Novo Item!* se o usuário seleciona o novo item. Como normal, a função *ProcJan* trata o processamento alterado dentro do arquivo *Mod_Menu.cpp*.

Você verá que o programa *Mod_Menu.cpp* usa a função *ModifyMenu* para alterar o valor da string do item de menu. Adicionalmente, o código verifica os identificadores de menu para garantir que a função capture o item recém-criado.

1307 USANDO ENABLEMENUITEM PARA CONTROLAR OS MENUS

Como você aprendeu, seus programas podem usar a função *ModifyMenu* para controlar a aparência dos itens dentro do menu. No entanto, como também foi visto, usar funções específicas para enfocar problemas específicos freqüentemente resulta na execução mais rápida do programa. Para lhe ajudar a habilitar, desabilitar ou acinzentar um item de menu que você especificar, seus programas podem usar a função *EnableMenuItem* em vez da função *ModifyMenu*. Você usará a função *EnableMenuItem* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL EnableMenuItem(
   HMENU hMenu,          // indicativo para menu
   UINT uIDEnableItem    // item de menu a habilitar, desabitiar ou acinzentar
   UINT uEnable          // sinalizadores de item de menu
);
```

O parâmetro *hMenu* é o indicativo para o menu que você quer modificar. O parâmetro *uIDEnableItem* especifica o item de menu que você quer habilitar, desabilitar ou acinzentar conforme o parâmetro *uEnable* determinar. O parâmetro *uIDEnableItem* especifica um item em uma barra de menu, menu ou submenu. O parâmetro *uEnable* especifica sinalizadores que controlam a interpretação do parâmetro *uIDEnableItem* e indica se o item de menu está em um estado habilitado, desabilitado ou acinzentado. O parâmetro *uEnable* precisa ser

uma combinação de *MF_BYCOMMAND* ou *MF_BYPOSITION* e *MF_ENABLED, MF_DISABLED,* ou *MF_GRAYED,* como mostra a Tabela 1307.

Tabela 1307 Os valores possíveis para o parâmetro ***uEnable***.

Sinalizadores	Descrição
MF_BYCOMMAND	Indica que *uIDEnableItem* dá o identificador do item de menu. Se o usuário não especifica o sinalizador *MF_BYCOMMAND* nem *MF_BYPOSITION*, o sinalizador *MF_BYCOMMAND* é o sinalizador padrão.
MF_BYPOSITION	Indica que *uIDEnableItem* dá a posição relativa com base em zero do item de menu.
MF_DISABLED	Indica que o item de menu está em um estado desabilitado, mas não cinza, de modo que o usuário não pode selecioná-lo.
MF_ENABLED	Indica que o Windows deve habilitar o item de menu e restaurá-lo a partir de um estado acinzentado para que o usuário possa selecioná-lo.
MF_GRAYED	Indica que o item de menu está em um estado acinzentado ou desabilitado para que o usuário não possa selecioná-lo.

Um aplicativo precisa usar o sinalizador *MF_BYPOSITION* para especificar o indicativo de menu correto. Caso seu programa especifique o indicativo de menu para a barra de menu, o Windows efetua a ação no item de menu de alto nível (um item na barra de menu). Para definir o estado de um item em um menu ou submenu suspenso por posição, um aplicativo precisa especificar o indicativo para o menu ou submenu suspenso.

Quando um aplicativo especifica o sinalizador *MF_BYCOMMAND*, o Windows verifica todos os itens que abrem submenus dentro do menu. Portanto, a não ser que itens de menus duplicados estejam presentes, especificar o indicativo de menu para a barra de menu é suficiente.

Para compreender melhor o processamento que *EnableMenuItem* executa, considere o programa *Enab_Dis.cpp*, contido no CD-ROM que acompanha este livro. O programa *Enab_Dis.cpp* usa a função *EnableMenuItem* para habilitar e desabilitar o item *IDM_ITEM1*. Como é normal, a função *ProcJan* dentro do arquivo *Enab_Dis.cpp* contém o código para efetuar as ações de habilitação e desabilitação, como mostrado aqui:

```
case IDM_TESTAR :
     {
        HMENU hMenu  = GetMenu(hJan);
        UINT uState = GetMenuState(hMenu, IDM_ITEM1, MF_BYCOMMAND);
        if (uState & MFS_GRAYED)
           EnableMenuItem(hMenu, IDM_ITEM1, MFS_ENABLED | MF_BYCOMMAND);
        else
           EnableMenuItem(hMenu, IDM_ITEM1, MFS_GRAYED | MF_BYCOMMAND);
     }
     break;
```

USANDO APPENDMENU PARA EXPANDIR UM MENU 1308

Como você aprendeu, seus programas podem executar muitas atividades diferentes nos menus após seus programas associarem o menu com a janela do programa. Uma das atividades mais comuns que seus programa executarão com um menu existente é anexar itens ao menu. A função *AppendMenu* anexará um novo item no final da barra de menu, menu suspenso, submenu, ou menu de atalho que você especificar. Você pode usar *AppendMenu* para especificar o conteúdo, aparência e comportamento do item de menu, como mostrado aqui:

```
BOOL AppendMenu(
   HMENU hMenu,     // indicativo do menu a ser alterado
   UINT  uFlags,    // sinalizadores de item de menu
   UINT  uIDNovoItem, // identificador de item de menu ou
                      // indicativo de menu suspenso ou submenu
  LPCTSTR lpNovoItem  // conteúdo do item de menu
);
```

Como é normal, o parâmetro *hMenu* identifica a barra de menu, o menu suspenso, o submenu, ou o menu de atalho que o *AppendMenu* deverá modificar. O parâmetro *uFlags* especifica sinalizadores para controlar a aparência e o comportamento do novo item de menu. O parâmetro *uIDNovoItem* especifica o identificador do novo item de menu ou, se o parâmetro *uFlags* está definido como *MF_POPUP*, o indicativo do menu ou do submenu suspenso. O parâmetro *lpNovoItem* especifica o conteúdo do novo item de menu. A interpretação da função *AppendMenu* de *lpNovoItem* depende se o parâmetro *uFlags* inclui o sinalizador *MF_BITMAP, MF_OWNERDRAW,* ou *MF_STRINGS,* como mostrado na Tabela 1306.3.

Como você aprendeu, o aplicativo precisa chamar a função *DrawMenuBar* sempre que um menu é modificado, independentemente se o menu está em uma janela atualmente mostrada. Seu programa pode definir diversos sinalizadores dentro do parâmetro *uFlag,* como detalha a Tabela 1308.

Tabela 1308 Os valores possíveis para o parâmetro ***uFlags***.

Valor	Descrição
MF_BITMAP	Usa um mapa de bits como o item de menu. O parâmetro *lpNovoItem* contém o indicativo para o mapa de bits.
MF_CHECKED	Coloca uma marca de verificação ao lado do item de menu. Se o aplicativo fornece mapas de bits de marcas de verificação, esse sinalizador exibirá uma marca de verificação ao lado do item de menu.
MF_DISABLED	Desabilita o item de menu para que o usuário não possa selecioná-lo, e restaura o item de menu de seu estado cinza.
MF_ENABLED	Habilita o item de menu para que o usuário não possa selecioná-lo, mas o sinalizador não o torna acinzentado.
MF_GRAYED	Desabilita o item de menu e o torna acinzentado para que o usuário não possa selecioná-lo.
MF_MENUBARBREAK	Funciona da mesma forma que o sinalizador *FM_MENUBREAK* para uma barra de menu. Para um menu suspenso, submenu, ou menu de atalho, uma linha vertical separa a nova coluna da antiga.
MF_MENUBREAK	Coloca o item em uma nova linha (para uma barra de menu) ou em uma nova coluna (para um menu suspenso, submenu ou menu de atalho) sem separar as colunas.
MF_OWNERDRAW	Especifica que o item é um item desenhado pelo usuário. Antes de o programa exibir o menu pela primeira vez, a janela que possui o menu recebe uma mensagem *WM_MEASUREITEM* para recuperar a largura e a altura do item de menu. O Windows então envia a mensagem *WM_DRAWITEM* para o procedimento de janela da janela proprietária sempre que você precisar atualizar a aparência do item de menu.
MF_POPUP	Especifica que o item de menu abre um menu suspenso ou um submenu. O parâmetro *uIDNovoItem* especifica o indicativo para o menu ou submenu suspenso. Você usará o sinalizador *MF_POPUP* para acrescentar um nome de menu a uma barra de menu, ou um item de menu que abre um submenu para um menu suspenso, submenu, ou menu de atalho.
MF_SEPARATOR	Desenha uma linha divisória horizontal. Você usará o sinalizador *MF_SEPARATOR* em um menu suspenso, submenu, ou menu de atalho. Seus programas não podem desabilitar, acinzentar ou destacar a linha. A função *AppendMenu* ignora os parâmetros *lpNovoItem* e *uIDNovoItem* se você especificar o tipo *MF_SEPARATOR*.
MF_STRING	Especifica que o item de menu é uma string de texto; o parâmetro *lpNovoItem* aponta para a string.
MF_UNCHECKED	Não coloca uma marca de verificação ao lado do item (padrão). Se o aplicativo fornece mapas de bits (veja *SetMenuItemBitmaps*), este sinalizador exibe o mapa de bits não-assinalado ao lado do item de menu.

Nota: O Windows não lhe permitirá usar os seguintes grupos de sinalizadores juntos quando você chamar a função AppendMenu:

- *MF_BYCOMMAND* e *MF_BYPOSITION*
- *MF_DISABLED, MF_ENABLED* e *MF_GRAYED*
- *MF_BITMAP, MF_STRING, MF_OWNERDRAW* e *MF_SEPARATOR*
- *MF_MENUBARBREAK* e *MF_MENUBREAK*
- *MF_CHECKED* e *MF_UNCHECKED*

Para compreender melhor o processamento executado pela função *AppendMenu*, considere o programa *Adi_Novo cpp* contido no CD-ROM que acompanha este livro, que acrescenta o item de menu New Item ao menu em sua própria linha toda vez que o usuário seleciona o item de menu *Testar!*. A função *ProcJan* captura a seleção *Testar!* e usa *AppendMenu* para acrescentar um novo item, como mostrado aqui:

```
case IDM_TESTAR :
     // Acrescenta nova opção de menu em uma nova linha.
     AppendMenu(GetMenu(hJan), MFT_STRING | MFT_MENUBARBREAK, 120,
     "Novo Item");
     DrawMenuBar(hJan);
     break;
```

USANDO DELETEMENU PARA EXCLUIR SELEÇÕES DO MENU 1309

Na dica anterior, você usou a função *AppendMenu* para acrescentar itens a um menu especificado. Você também poderá usar a função *DeleteMenu* nos seus programas para excluir um item do menu que você especificar. Se o item de menu abre um menu ou um submenu, a função *DeleteMenu* destrói o indicativo para o menu ou submenu e libera a memória que o menu ou submenu estava usando. Você implementará a função *DeleteMenu* dentro de seus programas, como mostrado aqui:

```
BOOL DeleteMenu(
   HMENU hMenu,       // indicativo de menu
   UINT uPosicao,     // identificador de item de menu ou de posição
   UINT uFlags        // sinalizador de item de menu
);
```

Como sempre, o aplicativo precisa chamar a função *DrawMenuBar* sempre que modifica um menu, esteja ou não o menu em uma janela que está sendo mostrada. Para compreender melhor o processamento que o comando *DeleteMenu* executa, considere o programa *Adi_Remo cpp* contido no CD-ROM que acompanha este livro, o qual acrescenta três novos itens a um menu na criação da janela, depois exclui esses novos itens um de cada vez quando o usuário seleciona um item. O programa trata o processamento dentro das opções *WM_CREATE* e *WM_COMMAND* da função *ProcJan*, como mostrado aqui:

```
case WM_CREATE :
   {
     HMENU hMenu = GetMenu(hJan);

     AppendMenu(hMenu, MFT_STRING, IDM_ITEM1, "Item&1");
     AppendMenu(hMenu, MFT_STRING, IDM_ITEM2, "Item&2");
     AppendMenu(hMenu, MFT_STRING, IDM_ITEM3, "Item&3");
   }
   break;

case WM_COMMAND :
   switch(LOWORD(wParam))
   {
     case IDM_ITEM1 :
     case IDM_ITEM2 :
     case IDM_ITEM3 :
       {
         HMENU hMenu = GetMenu(hJan);
```

```
            DeleteMenu(hMenu,LOWORD(wParam),MF_BYCOMMAND);
            DrawMenuBar(hJan);
         }
      break;
```

1310 USANDO TECLAS ACELERADORAS COM ITENS DE MENU

Como você aprendeu, seus programas podem usar (&) dentro de uma definição de menu para definir a tecla aceleradora de um item de menu. No entanto, seus programas também poderão acrescentar aceleradores à definição de um menu com novos itens de menu quando seu programa estiver em execução. Para fazer isso, seus programas usarão a função *CreateAcceleratorTable*, que criará uma tabela aceleradora. Você implementará a função *CreateAcceleratorTable* dentro de seus programas de acordo com o seguinte protótipo:

```
HACCEL CreateAcceleratorTable(
   LPACCEL lpacel,   // ponteiro para matriz de estrutura
                     // com dados aceleradores
   int cEntradas     // número de estruturas na matriz
);
```

O parâmetro *lpacel* aponta para uma matriz de estruturas ACCEL que descreve a tabela aceleradora. O parâmetro *cEntradas* especifica o número de estruturas *ACCEL* na matriz. Cada elemento da estrutura *ACCEL* dentro da matriz define, dentro de uma tabela aceleradora, uma tecla aceleradora que seu programa usará. A API Win32 define a estrutura *ACCEL* como segue:

```
typedef struct tagACCEL {  // acel
   BYTE   fVirt;
   WORD   key;
   WORD   cmd;
 } ACCEL;
```

A Tabela 1310.1 detalha os membros da estrutura *ACCEL*.

Tabela 1310.1 Os membros da estrutura *ACCEL*.

Membros	Descrição
fVirt	Especifica os sinalizadores aceleradores. Este membro pode ser uma combinação dos valores detalhados na Tabela 1310.2
key	Especifica a tecla aceleradora. Esse membro pode ser um código de tecla virtual ou um código de caractere ASCII.
cmd	Especifica o identificador do acelerador. O Windows coloca o valor *cmd* na palavra de baixa ordem do parâmetro *wParam* da mensagem *WM_COMMAND* ou *WM_SYSCOMMAND* quando o usuário pressiona a tecla aceleradora.

Como indica a Tabela 1310.1, o membro *fVirt* precisa ser um ou mais dos valores listados na Tabela 1310.2

Tabela 1310.2 Os valores possíveis para o membro *fVirt*.

Valor	Significado
FALT	O usuário precisa manter pressionada a tecla *Alt* enquanto pressiona a tecla aceleradora.
FCONTROL	O usuário precisa manter pressionada a tecla Ctrl enquanto pressiona a tecla aceleradora.
FNOINVERT	Especifica que o Windows destacará um item de menu de alto nível enquanto o usuário pressiona a tecla aceleradora. Se o usuário não especificar o sinalizador *FNOINVERT*, o Windows destacará um item de menu de alto nível, se possível, sempre que o usuário pressionar a tecla aceleradora.
FSHIFT	O usuário precisa manter a tecla Shift pressionada ao pressionar a tecla aceleradora.
FVIRTKEY	O membro da tecla especifica um código de tecla virtual. Se você não especificar esse sinalizador, o sistema assumirá que a tecla especifica um código de caractere ASCII.

Para compreender melhor o processamento que a função *CreateAcceleratorTable* executa, você criará uma função na próxima dica que cria uma nova tabela aceleradora. No entanto, se seus programas irão usar tabelas aceleradoras, você também precisará modificar a rotina de tratamento de mensagem. Por exemplo, o código a seguir mostra uma modificação simples no laço de mensagem que agora permite que o programa suporte tabelas aceleradoras:

```
while(GetMessage(&msg, NULL, 0, 0)
   {
      if (!hAccel || !TranslateAccelerator(hJan, hAccel, &msg))
      {
          TranslateMessage(&msg);
          DispatchMessage(&msg);
      }
   }
```

A variável *hAccel* é uma variável global com o valor inicial de *NULL* que, mais tarde, conterá o indicativo para a tabela aceleradora. A função *TranslateAccelerator* traduz qualquer mensagem *AcceleratorTable* à combinação *WM_COMMAND* e de constante de menu. Observe que o programa verifica a função *TranslateAccelerator*. Se *TranslateAccelerator* tratar a mensagem, o laço não deverá permitir o processamento normal seguir, pois *TranslateAccelerator* chamará a função *ProcJan* automaticamente.

Nota: Antes de um aplicativo terminar, ele precisa usar a função DestroyAcceleratorTable para destruir cada tabela aceleradora que usou a função CreateAcceleratorTable para criar.

CRIANDO UMA TABELA ACELERADORA DE EXEMPLO 1311

Como você aprendeu na dica anterior, seus programas podem criar tabelas aceleradoras durante a execução para fornecer funcionalidade adicional para os menus que você criar ou modificar dinamicamente. Você também aprendeu que a função *CreateAccelerator* usa uma matriz de estruturas *ACCEL* para criar a tabela aceleradora. Dentro de seus programas, você primeiro criará a matriz, e, depois, chamará a função *CreateAccelerator*. Para compreender melhor o processo de criar a tabela aceleradora, considere a seguinte função *AdicNovoItemTestar* a partir do programa *Cria_Ace.cpp* contido no CD-ROM que acompanha este livro, que cria uma nova entrada de tabela aceleradora cada vez que o usuário seleciona o item de menu *Testar!*:

```
#define IDM_NOVABASE 300

VOID AdicNovoItemTestar(HWND hJan)
 {
   static int nNum = 0;

   char   szMenuItem[20];

   HMENU  hMenu       = GetMenu(hJan);
   ACCEL* pDadosAcel  = NULL;
   ACCEL* pAcelAtual  = NULL;
   HANDLE hDadosAcel  = NULL;
   int    nNumAcel    = 1;

   if (nNum == 4)    // Máximo de 4 novos itens permitidos.
      return;

   // Se a tabela aceleradora existe, pega número de itens.
   if (hAccel)
      nNumAcel = CopyAcceleratorTable(hAccel, NULL,0)+1;

   // Aloca uma matriz de estruturas ACCEL.
   hDadosAcel = GlobalAlloc(GHND, sizeof(ACCEL) * nNumAcel);
   if (hDadosAcel)
      pDadosAcel = (ACCEL*)GlobalLock(hDadosAcel);
```

```
    // Se existe uma tabela aceleradora, copia os itens
    // na matriz recém-alocada.
    if (hAccel && pDadosAcel)
    {
       CopyAcceleratorTable(hAccel, pDadosAcel, nNumAcel - 1);

       DestroyAcceleratorTable(hAccel);
       hAccel = NULL;
    }

    // Acrescenta a nova opção do menu e a tecla aceleradora
    if (pDadosAcel)
    {
       // Obtém um ponteiro para a nova tecla aceleradora na matriz.
       pAcelAtual = (ACCEL*)(pDadosAcel+nNumAcel-1);

       // Cria uma nova opção de menu no menu.
       nNum++;
       wsprintf(szMenuItem, "Novo Item&%d", nNum);
       AppendMenu(hMenu, MFT_STRING, IDM_NOVABASE+nNum, szMenuItem);
       DrawMenuBar(hJan);

       // Define uma nova aceleradora de F1,F2,F3, ou F4
       // para a nova opção de menu.
       pAcelAtual->fVirt = FNOINVERT | FVIRTKEY;
       pAcelAtual->cmd   = IDM_NOVABASE+nNum;
       pAcelAtual->key   = (nNum == 1 ? VK_F1 :
                            nNum == 2 ? VK_F2 :
                            nNum == 3 ? VK_F3 :
                            /* padrão */ VK_F4);

       // Cria a nova tabela aceleradora.
       hAccel=CreateAcceleratorTable(pDadosAcel,nNumAcel);
       GlobalUnlock(hDadosAcel);
    }

    if (hDadosAcel)
       GlobalFree(hDadosAcel);
}
```

A função *AdicNovoItemTestar* executa um longo processamento, a maior parte do qual é explicado pelos comentários dentro do código. No entanto, vale a pena observar a criação do elemento da estrutura para a nova tecla aceleradora. Neste exemplo em particular, não mais de quatro teclas de função podem servir como atalhos; no entanto, o limite poderia também ser 10, ou o usuário poderia determinar as teclas de atalho.

1312 COMPREENDENDO MELHOR A ESTRUTURA DO ARQUIVO DE RECURSO

Como você aprendeu, virtualmente todo programa usa recursos. Para ajudar a manter esses recursos organizados e facilmente acessíveis, você geralmente armazenará informações sobre os recursos que um programa usa dentro de um arquivo de recurso. Os arquivos de recurso também são eficientes porque o programa normalmente carrega recursos na memória apenas quando o programa precisa do recurso.

Tipicamente, um compilador de recursos (em geral chamado *rc.exe*) compila o arquivo de recurso em um arquivo *RES*. O linkeditor então vincula o arquivo *RES* ao final do arquivo de programa executável totalmente executavel. Todos os recursos que o arquivo de recurso define então se tornam disponíveis para a unidade para uso durante a execução.

O arquivo de recurso pode incluir cinco tipos de script de recurso de uma única linha: *BITMAP, CURSOR, ICON, FONT* e *MESSAGETABLE*. Cada um desses comandos carrega um arquivo de dados do tipo que seu nome especifica na tabela de recurso. Após você incluir os recursos dentro do arquivo de recurso, seu programa poderá, depois, usar funções *Load*, tais como *LoadIcon*, para acessar esses itens. Você tipicamente implementará um tipo de recurso de uma única linha como mostrado aqui:

```
MEUAPL  ICON    DISCARDABLE     "GENERICO.ICO"
```

Além dos cinco tipos de script de recursos de uma única linha, existem cinco tipos de scripts de recursos de múltiplas linhas: *ACCELERATOR, DIALOG, MENU, RCDATA* e *STRINGTABLE*. Você também já aprendeu sobre os tipos *ACCELERATOR* e *MENU*. Você aprenderá mais sobre o tipo *DIALOG* a partir da Dica 1319, mais à fente. As próximas seis dicas explicam como você usará o tipo *STRINGTABLE* e *RCDATA*.

Os tipos de arquivo de recurso de múltiplas linhas são relativamente fáceis de identificar. Cada recurso de múltipla linha inclui um comando de tipo, um bloco *BEGIN-END* e comandos dentro do bloco *BEGIN-END* que podem incluir blocos *BEGIN-END* adicionais, como mostrado aqui:

```
NOVOMENU    MENU    DISCARDABLE
BEGIN
  POPUP "&Arquivo"
  BEGIN
        MENUITEM "Sai&r",       IDM_SAIR
  END
  MENUITEM "&Olá pessoal!",   IDM_ANTIGO
```

APRESENTANDO AS TABELAS DE STRING 1313

Na dica anterior você viu que, um dos tipos de múltiplas linhas que os arquivos de recurso suportam são o tipo *STRINGTABLE*. A maioria dos aplicativos usa uma série de strings de caracteres na escrita de mensagens e de caracteres. O Windows fornece tabelas de string como uma alternativa ao método convencional de colocar strings na área de dados estáticos do programa. Seus programas podem, portanto, definir strings de caracteres dentro do arquivo de recurso e dar à string um valor de identificação, como mostrado aqui:

```
STRINGTABLE
BEGIN
  IDS_STRING1  "Exemplo Simples de String."
  IDS_STRING2  "Bíblia do Programador C/C++"
  IDS_STRING3  "Jamsa e Klander"
END
```

Além da definição de seu conteúdo dentro do recurso, você tipicamente definirá os valores de identificação da string (tal como *IDS_STRINGS3)* em um arquivo de cabeçalho separado que você então incluirá dentro o arquivo de recurso e o módulo (ou módulos) que você acessará as strings. Quando o aplicativo precisar acessar os dados, você usará a função *LoadString* para copiar os dados do caractere do arquivo de recurso em um buffer da memória. As strings em um tabela de recursos podem conter caracteres de controle (tais como tabs e avanço de linha), bem como caracteres de impressão normais.

Existem diversas vantagens significativas de programação para usar as tabelas de string. A principal vantagem na programação do uso das tabelas de string é a redução no uso de memória do seu programa. Como seu programa não pode carregar as strings até que ele precise delas, não é necessário armazenar as strings dentro da área de dados estáticos do seu programa. Por essa razão, você deverá evitar copiar os dados da tabela de string em um buffer de memória estática, pois, fazer isso, afetará o propósito da tabela de string. Em seu lugar, você deverá copiar a tabela string em uma variável local (uma variável na pilha) ou na memória globalmente alocada.

Outra vantagem significativa do uso das tabelas de string é o seu suporte para múltiplas linguagens. A API Win32 suporta os recursos de múltiplos idiomas dentro de um único aplicativo — o que significa que você pode distribuir o mesmo programa executável em vários países sem ter que modificá-lo. Dicas posteriores usarão tabelas de string para manter as informações das janelas.

1314 COMPREENDENDO OS RECURSOS PERSONALIZADOS

Como você aprendeu na Dica 1312, um dos tipos de recursos de múltiplas linhas é o tipo *RCDATA*. Você pode usar o tipo *RCDATA* para armazenar outros tipos de dados estáticos, especificamente dados binários crus. Por exemplo, a listagem de código a seguir armazena diversos dados de diferentes tipos dentro do recurso *DataID:*

```
DataID RCDATA
BEGIN
   3
   40
   0x8232
   "Dados string (continuação) ... "
   "Mais Dados da String\0"
END
```

Embora você possa incluir dado de recurso comum diretamente dentro do arquivo de programa, ou lê-lo a partir de um arquivo de dado externo, o melhor lugar para armazenar o dado de recurso personalizado é dentro de um arquivo externo. O compilador de recurso pode então acrescentar o conteúdo do arquivo externo ao dado de recurso quando compila o arquivo de recurso. Por exemplo, os dois comandos a seguir definem os tipos de recursos personalizados *TEXT* e *METAFILE*, atribui um identificador aos tipos, e, depois acrescenta os recursos ao arquivo de recurso:

```
happy    TEXT      "happycao.txt"
foto     METAFILE  "happyfot.wmf"
```

Quando você definir os tipos de recursos personalizados, usará a função *FindResource* junto com a função *LoadResource* para carregar os recursos personalizados nos seus programas. A Dica 1318, à frente, explicará a função *LoadResource*.

1315 CARREGANDO TABELAS DE STRING NOS PROGRAMAS COM LOADSTRING

Você já sabe que, pode criar tabelas de string dentro de seus arquivos de recurso. Agora atribua a cada string dentro de uma tabela de string um valor de identificação específico, que normalmente você definirá dentro do arquivo de cabeçalho do programa, como mostrado aqui:

```
#define IDM_SAIR   100
#define IDM_TESTAR 200
#define IDM_SOBRE  301
```

Após você definir uma tabela de string, seus programas usarão a função *LoadString* para carregar um recurso string a partir do arquivo executável associado com um módulo especificado, copiarão a string em um buffer e anexarão um caractere *NULL* finalizador ao buffer. Você usará a função *LoadString* dentro de seus programas, como mostrado no protótipo a seguir:

```
int LoadString(
   HINSTANCE hCopia,     // indicativo do módulo que contém o recurso string
   UINT uID,             // identificador do recurso
   LPTSTR lpBuffer,      // endereço do buffer para recursos
   int nBufferMax        // tamanho do buffer
);
```

O parâmetro *hCopia* identifica uma ocorrência do módulo cujo arquivo executável contém o recurso string. O parâmetro *uID* especifica o identificador inteiro da string que *LoadString* deverá carregar. O parâmetro *lpBuffer* aponta para o buffer que receberá a string. Finalmente, o parâmetro *nBufferMax* especifica o tamanho do buffer em bytes (versão ANSI) ou caracteres (versao Unicode). A função trunca e finaliza a string com *NULL* se ela é maior que o número de caracteres que você especifica.

Se *LoadString* for bem-sucedida, ela retornará o número de bytes (versão ANSI) ou caractere (versão Unicode) que copiou para o buffer, não incluindo o caractere terminado por *NULL*, ou zero se o recurso string não existe. Para obter informações estendidas de erro, você pode chamar a função *GetLastError*. Para compreender melhor o processamento efetuado por *LoadString*, considere o seguinte fragmento de código do programa *Carrega.cpp* contido no CD-ROM que acompanha este livro:

```
case IDM_TESTAR:
  {
    char szString[40];
    SHORT idx;

    for (idx = IDS_STRINGBASE; idx < IDS+STRINGBASE+3; idx++)
      {
        LoadString(hCop, idx, szString, 40);
         MessageBox(hjan, szString, "String Carregada",
                    MB OK | MB ICONINFORMATION);
      }
  }
  break;
```

O programa *Carrega.cpp* carregará e exibirá três strings separadas quando o usuário selecionar a opção *Testar!* O programa exibirá cada string dentro de uma caixa de mensagem.

LISTANDO O CONTEÚDO DE UM ARQUIVO DE RECURSO 1316

Como você aprendeu, pode armazenar muitos tipos diferentes de dados personalizados dentro de um arquivo de recurso. A função *EnumResourceNames* pesquisa um módulo para cada recurso do tipo que você especificar dentro do parâmetro *lpszTipo*, e passa o nome de cada recurso que localizar para uma função de callback definida pelo aplicativo. A função *EnumResourceNames* continua enumerando os nomes de recursos até que a função de callback retorne *False* ou até que *EnumResourceNames* tenha enumerado todos os nomes de recursos. Você usará a função *EnumResourceNames* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL EnumResourceNames(
   HINSTANCE hModulo,              // tratamento do módulo de recursos
   LPCTSTR  lpszTipo,              // ponteiro para tipo de recurso
   ENUMRESNAMEPROC lpEnumFunc,     // ponteiro para função de callback
   LONG lParam                     // parâmetro definido pelo aplicativo
);
```

A função *EnumResourceNames* aceita os parâmetros detalhados na Tabela 1316.1.

Tabela 1316.1 Os parâmetros para a função ***EnumResourceNames***.

Parâmetros	Descrição
hModulo	Identifica o módulo cujo arquivo executável contém os recursos para os quais *EnumResourceNames* deve enumerar os nomes. Se esse parâmetro for *NULL*, a função irá enumerar os nomes dos recursos no módulo usado para criar o processo atual.
lpszTipo	Aponta para uma string terminada por *NULL* que especifica o nome do tipo do recurso para o qual *EnumResourceNames* está enumerando o nome. Para os tipos de recursos padrão, esse parâmetro pode ser um dos valores detalhados na Tabela 1316.2
lpEnumFunc	Aponta para a função de callback que *EnumResourceNames* deve chamar para cada nome de recurso enumerado.
lParam	Especifica um valor definido pelo aplicativo que *EnumResourceNames* passa para a função de callback. Seus programas poderão usar esse parâmetro quando estiverem checando os erros.

Como observa a Tabela 1316.1, o parâmetro *lpszTipo* pode ter um dentre vários valores, como detalha a Tabela 1316.2.

Tabela 1316.2 Os valores possíveis para ***lpszTipo***.

Valor	Significado
RT_ACCELERATOR	Tabela aceleradora
RT_ANICURSOR	Cursor animado
RT_ANIICON	Ícone animado
RT_BITMAP	Recurso de mapa de bits
RT_CURSOR	Recurso de cursor dependente do hardware
RT_DIALOG	Caixa de diálogo
RT_FONT	Recurso de fonte
RT_FONTDIR	Recurso de diretório de fonte
RT GROUP_CURSOR	Recurso de cursor independente do hardware
RT_GROUP_ICON	Recurso de ícone independente do hardware
RT_ICON	Recurso de ícone dependente do hardware
RT_MENU	Recurso de menu
RT_MESSAGETABLE	Entrada na tabela de mensagens
RT_PLUGPLAY	Recurso de Plug and Play
RT RCDATA	Recurso definido pelo aplicativo (dados crus)
RT_STRING	Entrada em tabela de string
RT_VERSION	Recurso de versão
RT_VXD	VXD (Controlador de Dispositivo Virtual)

Para compreender melhor como a função *EnumResourceNames* funciona, considere o programa *3_figs.cpp*, contido no CD-ROM que acompanha este livro. O arquivo de recurso para o programa *3_figs.cpp* inclui três mapas de bits. O programa usa a função *EnumResourceNames* e uma função de callback chamada *PaintBitmaps* para enumerar os recursos do tipo RT_BITMAP dentro de um arquivo de recursos *3_figs.cpp*. A listagem de código a seguir mostra a função de callback *PaintBitmaps:*

```
BOOL CALLBACK PaintBitmaps(HANDLE hModulo, LPCTSTR lpszTipo, LPCTSTR lpszNome,
                           LONG lParam)
{
  HBITMAP hBitmap = LoadBitmap(hModulo, lpszNome);

  if (hBitmap)
  {
     BITMAP bm;
     HDC    hMemDC;
     HWND   hJan = (HWND)lParam;
     HDC    hDC  = GetDC(hJan);
     HFONT  hFonteAntiga;

     // Pega o tamanho do mapa de bits.
     GetObject(hBitmap, sizeof(BITMAP), &bm);

     // Cria um DC em memória para selecionar o mapa de bits
     hMemDC = CreateCompatibleDC(hDC);
     SelectObject(hMemDC, hBitmap);

     // Exibe o mapa de bits, ampliando-o para 50X50 pixels.
     StretchBlt(hDC, gnPos, 0, 50, 50,  hMemDC, 0, 0, bm.bmWidth,
```

```
            bm.bmHeight, SRCCOPY);
            // Exibe o nome do mapa de bits.
            hFonteAntiga = SelectObject(hDC, GetStockObject(ANSI_VAR_FONT));
            TextOut(hDC, gnPos, 60, lpszNome, strlen(lpszNome));
            SelectObject(hDC, hFonteAntiga);
            DeleteDC(hMemDC);
            ReleaseDC(hJan, hDC);
            DeleteObject(hBitmap);
            gnPos += 100;
        }

        return(TRUE);
    }
```

A função de callback *PaintBitmaps* efetua o processamento significativo, grande parte do qual será explicado em detalhes em dicas posteriores. A função de callback pinta cada mapa de bits que *EnumResourceTypes* enumera na tela.

USANDO ENUMRESOURCETYPES COM ARQUIVOS DE RECURSOS 1317

Na dica anterior, você aprendeu sobre a função *EnumResourceNames*, que seus programas podem usar para enumerar os diferentes recursos de um determinado tipo dentro de um arquivo de recurso. No entanto, algumas vezes seus programas não saberão de antemão os diferentes tipos que o arquivo de recurso contém. Nesses casos, seus programas também podem usar a função *EnumResourceTypes*. A função *EnumResourceTypes* pesquisa os recursos em um módulo e passa cada tipo de recurso que encontrar a uma função de callback definida pelo aplicativo. A função *EnumResourceTypes* continua a enumerar os tipos de recursos até que a função de callback retorne *False* ou até que ela tenha enumerado todos os tipos de recursos. Você implementará a função *EnumResourceTypes* dentro de seus programas como mostrado aqui:

```
BOOL EnumResourceTypes(
    HMODULE hModulo,                // indicativo de módulo de recurso
    ENUMRESTYPEPROC lpEnumFunc,     // ponteiro para função de callback
    LONG lParam                     // parâmetro definido pelo aplicativo
);
```

Exatamente como a função *EnumResourceNames* usa uma função de callback, assim também acontece com a função *EnumResourceTypes*. O protótipo da função de *callback* que você usará com a função *EnumResourceTypes* é mostrado aqui (o nome da função *EnumResTypeProc* é um marcador de lugar para o nome de função definido pelo aplicativo ou definido pela biblioteca):

```
BOOL CALLBACK EnumResTypeProc(
    HANDLE  hModulo,      // indicativo de módulo de recurso
    LPTSTR lpszTipo,      // ponteiro para tipo de recurso
    LONG lParam           // parâmetro definido pelo aplicativo
);
```

O parâmetro *lpszTipo* aponta para uma string terminada por *NULL* que especifica o nome do tipo do recurso para o qual a função está enumerando o tipo. Para tipos de recursos padrão, o parâmetro *lpszTipo* pode ser um dos valores detalhados na Tabela 1316.2.

Para compreender melhor o processamento que a função *EnumResourceTypes* executa, considere o programa *EnumResT.cpp*, contido no CD-ROM que acompanha este livro. O programa *EnumResT.cpp* preenche uma caixa de lista com todos os tipos diferentes que enumera dentro de um determinado arquivo de recurso. O fragmento de código a seguir detalha a função de callback *ListResourceTypes*, que a função *EnumResourceTypes* usa como sua função de callback:

```
BOOL CALLBACK ListResourceTypes(HANDLE hModulo, LPTSTR lpszTipo, LONG lParam)
 {
   LPTSTR lpSomaString = lpszTipo;
   HWND   hCaixaLista  = (HWND)lParam;

   // Verifica se o tipo de recurso é um tipo predefinido.
   // Se for, define lpSomaString como uma string descritiva.
   switch(LOWORD(lpszTipo))
   {
      case RT_ACCELERATOR    : lpSomaString = "Accelerator"; break;
      case RT_BITMAP         : lpSomaString = "Bitmap"; break;
      case RT_DIALOG         : lpSomaString = "Dialog"; break;
      case RT_FONT           : lpSomaString = "Font"; break;
      case RT_FONTDIR        : lpSomaString = "FontDir"; break;
      case RT_MENU           : lpSomaString = "Menu"; break;
      case RT_RCDATA         : lpSomaString = "RC Data"; break;
      case RT_STRING         : lpSomaString = "String Table"; break;
      case RT_MESSAGETABLE   : lpSomaString = "Message Table"; break;
      case RT_CURSOR         : lpSomaString = "Cursor"; break;
      case RT_GROUP_CURSOR   : lpSomaString = "Cursor de Grupo"; break;
      case RT_ICON           : lpSomaString = "Ícone"; break;
      case RT_GROUP_ICON     : lpSomaString = "Ícone de Grupo"; break;
      case RT_VERSION        : lpSomaString = "Informações da Versão "; break;
   }

   SendMessage(hCaixaLista, LB_INSERTSTRING, (WPARAM)-1, (LPARAM)lpSomaString);
   return(TRUE);
 }
```

A função de callback recebe o valor *lpszTipo* da função *EnumResourceTypes* e converte o valor para um valor string mais reconhecível para o usuário, tal como "Ícone" ou "Menu".

1318 Carregando Recursos em Programas com FindResource

Como você aprendeu, seus programas podem definir qualquer quantidade de recursos personalizados dentro do arquivo de recurso. Em dicas anteriores, você usou as funções *EnumResourceNames* e *EnumResourceTypes* para listar todos os nomes de recurso de um determinado tipo e todos os tipos de recursos dentro de um determinado arquivo de recurso. Alternativamente, seus programas podem usar as funções *FindResource* e *LoadResource* para fazer praticamente a mesma tarefa. A função *FindResource* determina a localização de um recurso com o tipo e o nome que você especificar no módulo que você especificar. Você usará a função *FindResource* dentro de seus programas, como mostrado no protótipo a seguir:

```
HRSRC FindResource(
   HMODULE hModulo,     // indicativo de módulo de recurso
   LPCTSTR lpNome,      // ponteiro para nome de recurso
   LPCTSTR lpTipo       // ponteiro para tipo de recurso veja a Tabela 1316.2
);
```

Se a função *FindResource* for bem-sucedida, o valor de retorno será um indicativo para o bloco de informações do recurso especificado. Para obter um indicativo para o recurso, passe o indicativo que *FindResource* retorna para a função *LoadResource*. Se a função falhar, o valor de retorno será *NULL*.

Se a palavra mais significativa dos parâmetros *lpNome* e *lpTipo* for zero, a palavra menos significativa especificará o identificador inteiro do nome ou do tipo do recurso determinado. Caso contrário, esses parâmetros serão ponteiros *long* para strings terminadas por *NULL*. Se o primeiro caractere da string for um sinal de #, os caracteres restantes representarão um número decimal que especifica o identificador inteiro do nome ou tipo do recurso. Por exemplo, a string "#258" representa o identificador inteiro 258.

Seus aplicativos devem reduzir a quantidade de memória que os recursos requerem referenciando os recursos pelo identificador inteiro em vez do nome. Um aplicativo pode usar a função *FindResource* para localizar qualquer tipo de recurso, mas você deverá usar *FindResource* somente se o aplicativo precisar acessar os dados de recurso binário quando fizer chamadas subsequentes às funções *LoadLibrary* e *LockResource*. Para aprender mais sobre *LoadLibrary* e *LockResource*, consulte a documentação on-line do compilador.

Para usar um recurso imediatamente, um aplicativo deve usar uma das funções específicas de recursos detalhadas na Tabela 1318 para localizar e carregar os recursos em uma chamada.

Tabela 1318 As funções *Load* específicas dos recursos.

Função	Ação
FormatMessage	Carrega e formata uma entrada na tabela de mensagem
LoadAccelerators	Carrega uma tabela aceleradora
LoadBitmap	Carrega um recurso mapa de bits
LoadCursor	Carrega um recurso cursor
LoadIcon	Carrega um recurso ícone
LoadMenu	Carrega um recurso menu
LoadString	Carrega uma entrada na tabela de string

Por exemplo, um aplicativo pode usar a função *LoadIcon* para carregar um ícone para exibição na tela. No entanto, o aplicativo deverá usar *FindResource* e *LoadResource* se estiver carregando o ícone para copiar seus dados para outro aplicativo.

Para compreender melhor o processamento que *FindResource* executa, considere o programa *FindRes.cpp*, contido no CD-ROM que acompanha este livro. *FindRes.cpp* usa *FindResource* para carregar dados binários crus para dentro de uma estrutura. O programa *FindRes.cpp* tem modificações no arquivo de recursos, no arquivo de cabeçalho e no arquivo de programa em relação aos arquivos *genérico* originais. O arquivo de recursos inclui o seguinte código adicional, que simplesmente define um conjunto de valores hexadecimais:

```
TestData RCDATA  DISCARDABLE
BEGIN
   0x0001,
   0x0002,
   0x0003
END
```

O arquivo de cabeçalho define uma estrutura de dados, na qual o programa lerá os dados dentro do bloco *Testa!Dado*, como mostrado aqui:

```
typedef struct {
   SHORT Valor1;
   SHORT Valor2;
   SHORT Valor3;
 } RESDATA;
```

Finalmente, o arquivo de programa *FindRes.cpp* lê os dados na estrutura dentro da rotina de tratamento do item de menu *Testa!* na função *ProcJan*, como mostrado aqui:

```
case IDM_TESTAR :
   {
      HRSCR hres = FindResource(hCop, "TestaDado", RT_RCDATA);
```

```
    if (hres)
      {
        char szMsg[50];
        DWORD  tamanho = SizeofResource(hCop, hres);
        HGLOBAL hmem   = LoadResource(hCop, hres);
        RESDATA* pmem  = (RESDATA*) LockResource(hmem);
        wsprintf(szMsg, "Valores carregados: %d, %d, %d\nTamanho = %d",
        pmem->Valor1,
                pmem->Valor2, pmem->Valor3, tamanho);
        MessageBox(hjan, szMsg, lpszNomeApl, MB_OK);
      }
  }
  break;
```

Se o programa encontrar os dados do recurso abaixo da palavra-chave *Testa!Dado*, ele determinará a quantidade de dados e os salvará dentro da variável *tamanho*. O programa então usará o especificador *HGLOBAL* para alocar espaço no heap onde armazenar os dados. Finalmente, ele armazenará os dados dentro de uma ocorrência da estrutura *RESDATA*. Quando você compilar e executar o programa e selecionar o item de menu *Testa!*, a tela exibirá o valor dentro de uma caixa de mensagem.

1319 COMPREENDENDO AS CAIXAS DE DIÁLOGO

Nas dicas anteriores você criou diferentes janelas dentro de seus programas. No entanto, a maioria das janelas que você criou era de alto nível. Uma janela de alto nível processa mensagens de muitos tipos diferentes para executar o processamento do programa.

Uma *caixa de diálogo*, por outro lado é similar a uma janela instantânea. Uma caixa de diálogo recebe informações do usuário para alguma tarefa específica, tal como obter o nome de um arquivo ou uma string de caracteres para uma pesquisa. A principal diferença entre as caixas de diálogo e as janelas instantâneas é que as caixas de diálogo usam *gabaritos* que definem os controles que a caixa de diálogo exibe. Você usa o tipo de recurso *DIALOG* para definir esses gabaritos dentro do arquivo de recursos. Você também pode criar os gabaritos dinamicamente na memória durante a execução do programa.

As caixas de diálogo também funcionam diferentemente das janelas de ativação instantânea em que as caixas de diálogo usam uma função de processamento de mensagem padrão que interpreta as digitações, tais como as teclas de setas e TAB, o que permite que o usuário selecione mais facilmente os controles dentro da caixa de diálogo. Como uma regra, a função de processamento de mensagem adicional e a aparência da caixa de diálogo torna a caixa de diálogo mais conveniente para aceitar e processar a entrada simples do usuário. Nas dicas a seguir, você aprenderá sobre os tipos de caixa de diálogo e como usar as caixas de diálogo dentro de seus programas.

1320 DEFININDO OS TIPOS DE DIÁLOGO

Como você aprendeu na dica anterior, pode usar caixas de diálogo dentro de seus programas para fornecer e receber informações do usuário. Existem dois tipos básicos de caixas de diálogo: *modais* e *não-modais*.

Quando uma caixa de diálogo modal está na tela, o usuário não pode alternar para outra seção do aplicativo, a não ser que feche primeiro a caixa de diálogo. Por padrão, uma caixa de diálogo modal limita o acesso às outras janelas visíveis do aplicativo que chamou a caixa de diálogo. No entanto, o usuário ainda pode se alternar para outros aplicativos enquanto um simples aplicativo exibe a caixa de diálogo modal. A caixa de diálogo modal mais simples é a caixa de mensagem, que você usou em dicas anteriores.

Seus programas também podem especificar caixas de diálogo *modais do sistema*. Uma caixa de diálogo modal do sistema toma o controle de toda a tela e não permite que o usuário efetue processamento adicional em qualquer programa até que o usuário responda à caixa de diálogo. O uso de caixas de diálogo modais do sistema dentro de seus programas é apropriado somente no caso de um problema sério que o usuário não pode ignorar, tal como um erro do sistema. Para criar uma caixa de diálogo modal do sistema, você pode especificar o estilo *WS_SYSMODAL* no gabarito da caixa de diálogo ou usar a função *SetSysModalWindow* para criar a caixa de diálogo.

No entanto, tipicamente, você usará a função *DialogBox* para criar caixas de diálogo modais e modais do sistema. Quando seus programas usarem a função *DialogBox* para exibir uma caixa de diálogo, o Windows enviará todas as mensagens da janela chamadora para a caixa de diálogo.

Menos comum, mas ainda útil, são as caixas de diálogo não-modais. Ao contrário de uma caixa de diálogo modal, uma caixa de diálogo não-modal pode receber e perder a entrada (isto é, ela pode ser a janela ativa e também uma janela inativa). Uma caixa de diálogo não-modal tem uma duração indefinida. No entanto, como a caixa de diálogo não-modal pode existir por um período estendido de tempo, os programas que usam caixas de diálogo não-modais precisam garantir que o laço de mensagem compartilhem as mensagens com a caixa de diálogo. Você tipicamente construirá laços de mensagem para os programas que contêm caixas de diálogo não-modais, como mostrado aqui:

```
while (GetMessage(&msg, NULL, 0, 0))
 {
   if (hDlgNaoModal || !IsDialogMessage (hDlgNaoModal, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
 }
```

Na construção geral, o valor *hDlgNaoModal* representa um indicativo para a caixa de diálogo não-modal. Se a caixa de diálogo não-modal não estiver aberta no momento, o valor *hDlgNaoModal* precisará ser *NULL*. A função *IsDialogMessage* determina se uma mensagem do Windows é destinada para a caixa de diálogo. Em caso afirmativo, a função enviará a mensagem para o procedimento de tratamento de mensagem da caixa de diálogo, e, portanto, *TranslateMessage* e *DispatchMessage* não tratarão a mensagem.

USANDO O TECLADO COM CAIXAS DE DIÁLOGO 1321

Como você aprendeu, o Windows fornece lógica interna para tratar as caixas de diálogo na forma do procedimento especial de tratamento de mensagens de caixa de diálogo. A lógica interna inclui os meios de selecionar itens dentro da caixa de diálogo com digitações, em vez de usar o mouse. Seus programas podem fornecer três conjuntos de lógica de teclado: as *teclas de ativação* para selecionar itens usando combinações de tecla Alt+letra, resposta à tecla TAB para movimentos entre os controles e resposta para as teclas de setas para movimento entre e dentro dos controles.

Você fornecerá suporte para as combinações de tecla Alt+letra dentro de suas caixas de diálogo da mesma maneira que para os itens de menu. Dentro da string de texto do controle, preceda a letra que você quer que a combinação de tecla use com o sinal &. Por exemplo, a definição a seguir para um controle *DEFPUSHBUTTON* usará Alt+D como combinação de tecla que ativa o botão "Dálmata":

```
CONTROL "&Dálmata", IDC_DONE, "BUTTON",
   BS_DEFPUSHBUTTON | WS_TABSTOP | WS_CHILD, 45, 66, 48, 12
```

Algumas vezes, os usuários acham os controles de teclado mais convenientes que o mouse. No entanto, como com a maioria das outras simplificações para o usuário, tenha o cuidado de não definir um número excessivo de atalhos de teclado porque eles podem se tornar confusos em vez de úteis para os usuários. Para suportar os controles de teclado dentro de suas caixas de diálogo, você precisa definir certos elementos dentro do gabarito de caixa de diálogo, incluindo os estilos *WS_TABSTOP* e *WS_GROUP*.

O estilo *WS_TABSTOP* marca cada item que receberá o foco de entrada quando o usuário pressionar Tab ou Shift+Tab. O estilo *WS_GROUP* marca o início de um grupo. Todos os itens que você listar dentro do script de recursos até o próximo estilo *WS_GROUP* são parte de um único grupo. O usuário pode então usar as teclas de setas para percorrer os itens dentro de um grupo mas não pode usar as teclas de setas de um grupo para o outro.

COMPREENDENDO OS COMPONENTES DO GABARITO DA CAIXA DE DIÁLOGO 1322

Já sabemos que, dentro de um arquivo de recurso, você precederá uma série de recursos de controle de caixa de diálogo com um comando *DIALOG* para definir caixas de diálogo. A forma geral do gabarito de caixa de diálogo

é similar ao gabarito de recursos que você usou em dicas anteriores para projetar menus e itens de menu. A forma geral do gabarito de caixa de diálogo é mostrado aqui:

```
DBIdentifier   DIALOG  DISCARDABLE  Esq, Topo, Larg, Alt
STYLE  Estilo1 | Estilo2 | . . . | EstiloN
CAPTION "Título do Diálogo"
FONT Tamanho, "NomeFonte"
BEGIN
   TipoControle1 "Legenda do Controle", Ctrl ID,
                 Esq, Topo, Larg, Alt
   TipoControle2 "Legenda do Controle", Ctrl_ID,
                 Esq, Topo, Larg, Alt

   TipoControleN "Legenda do Controle", Ctrl_ID,
                 Esq, Topo, Larg, Alt
END
```

TipoControle refere-se a um de vários tipos de janela que você pode usar para criar controles em uma caixa de diálogo. A entrada *TipoControle* precisa ser um dos valores listados na Tabela 1322.

Tabela 1322 Valores possíveis para a entrada ***TipoControle*** dentro da definição da caixa de diálogo

Tipos de Controle Possíveis			
BUTTON	*CHECKBOX*	*COMBOBOX*	*CONTROL*
CTEXT	*DEFPUSHBUTTON*	*EDITTEXT*	*GROUPBOX*
ICON	*LISTBOX*	*LTEXT*	*PUSHBUTTON*
RADIOBUTTON	*RTEXT*	*SCROLLBAR*	*STATIC*

1323 CRIANDO UM GABARITO DE CAIXA DE DIÁLOGO ESPECÍFICO

Na dica anterior, você aprendeu a forma genérica da declaração de caixa de diálogo, que inclui muitos componentes importantes. Pode ser mais fácil para você compreender a definição de caixa de diálogo se você analisar uma declaração simples, como mostrado aqui:

```
TESTDIALOG DIALOG DISCARDABLE  20, 20, 180, 70
STYLE DS_MODALFRAME | WS_POPUP   WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Diálogo de Teste"
FONT 8, "MS Sans Serif"
BEGIN
    CHECKBOX        "Controle Caixa de Seleção", IDC_CHECKBOX,9,7,70,10
    GROUPBOX        "Botões de Opção",-1,7,21,86,39
    RADIOBUTTON     "Primeiro", IDC_RADIO1, 13, 32, 37, 10, WS_GROUP |
    WS_TABSTOP
    RADIOBUTTON     "Segundo",IDC_RADIO2, 13, 45, 39, 10
    PUSHBUTTON      "Concluído", IDCANCEL, 116, 8, 50, 14, WS GROUP
END
```

A declaração anterior cria a caixa de diálogo mostrada na Figura 1323.

Figura 1323 A caixa de diálogo *Teste de Diálogo*.

Quando você vir a caixa de diálogo resultante, compreender seus componentes dentro da definição do arquivo de recursos será significativamente mais fácil. Cada comando dentro do bloco *BEGIN-END* cria seu próprio controle. O primeiro comando cria a caixa de seleção no canto superior esquerdo do formulário. Ele também atribui o identificador *IDC_CHECKBOX* a esse controle para que a rotina de tratamento da mensagem possa responder apropriadamente às modificações do valor do controle (neste caso, se o controle está ou não atualmente marcado por uma marca de verificação).

O segundo comando cria a estrutura dos botões de opção, que o usuário não pode selecionar, de modo que a definição não lhe atribui um identificador. O terceiro e quarto comandos criam os botões de opção *Primeiro* e *Segundo* dentro da estrutura. Esses botões estão em um grupo; o usuário pode selecionar somente um botão de cada vez. Observe como o grupo inicia com o botão *IDC_RADIO1*, e o arquivo de recurso então declara outro grupo com o botão *Concluído*.

O último comando dentro do bloco usa o identificador *IDCANCEL* para criar o botão *Concluído*. Tipicamente, você atribuirá o identificador *IDCANCEL* a qualquer botão que feche a caixa de diálogo sem salvar as alterações em alguma outra posição.

COMPREENDENDO OS COMPONENTES DA DEFINIÇÃO DA CAIXA DE DIÁLOGO 1324

Cada definição de caixa de diálogo efetua processamento específico dentro de seu bloco *BEGIN-END* para criar os controles que a caixa de diálogo exibirá. No entanto, antes de especificar as definições do controle, a caixa de diálogo define seus próprios atributos padrão. Por exemplo, a caixa de diálogo *TESTADIALOGO* da dica anterior inicia com a seguinte definição de quatro linhas:

```
TESTADIALOGO DIALOG DISCARDABLE 20, 20, 180, 70
STYLE DS MODALFRAME | WS POPUP | WS_VISIBLE | WS_CAPTION   WS_SYSMENU
CAPTION "Diálogo de Teste"
FONT 8, "MS Sans Serif"
```

A primeira linha identifica a caixa de diálogo com o identificador *TESTADIALOGO*. A linha também declara as dimensões da caixa de diálogo. No exemplo anterior, a caixa de diálogo inicia em 20 unidades básicas de diálogo (DBU) para baixo e 20 unidades básicas de diálogo (DBU) para a esquerda a partir da borda do cliente da janela chamadora. A caixa de diálogo também tem 180 unidades básicas de diálogo (DBU) de largura e 70 unidades de altura.

A segunda linha define os estilos que a caixa de diálogo usará quando você a criar. Você pode usar todos os estilos de janela que a Dica 1272 definiu que iniciam com *WS* ou *DS* como o estilo da caixa de diálogo. Você sempre deve incluir o estilo *WS_VISIBLE* para tornar a caixa de diálogo visível. Você não pode estilizar as caixas de diálogo como *WS_MINIMIZEBOX* ou *WS_MAXIMIZEBOX*.

Você usará o especificador *CAPTION*, mostrado na terceira linha, com caixas de diálogo que você declara como tendo o estilo *WS_CAPTION*. Como regra, você deve legendar suas caixas de diálogo por duas razões. Primeira, ela lembra o usuário do propósito da caixa de diálogo. Segunda, ela permite que o usuário mova a caixa de diálogo pela tela. Para simplificar a interação do usuário com seus programas, certifique-se de colocar a string de título entre aspas.

Finalmente, o especificador *FONT* determina não somente a fonte que a caixa de diálogo usa, mas também o tamanho de todo controle dentro da caixa de diálogo e a própria caixa de diálogo. O especificador *FONT* tem um papel importante no tamanho da caixa de diálogo, pois o Windows calcula as unidades de base de diálogo (DBU) como frações do tamanho da fonte. Para a maioria das caixas de diálogo, uma fonte de 8 pontos, tal como MS Sans Serif, é uma boa escolha. Certifique-se de que o nome da fonte dentro das aspas é exatamente igual ao nome da fonte definido pelo sistema, ou o arquivo de recurso não será compilado corretamente.

DEFININDO OS CONTROLES DA CAIXA DE DIÁLOGO 1325

O Windows oferece dois modos equivalentes de definir os controles dentro de uma caixa de diálogo. Um modo de poder definir os controles é usar um comando explícito, tal como o comando *COMBOBOX*, como você viu na Dica 1323. O outro modo é usar o comando *CONTROL* e incluir o estilo de caixa de combinação dentro do controle como um parâmetro. Por exemplo, os dois comandos a seguir criam o mesmo controle:

```
CONTROL "Aperte se você for o Happy", IDC_BUTTON1, "button",
  BS_DEFPUSHBUTTON | WS_TABSTOP | WS_CHILD, 45, 66, 48, 12

DEFPUSHBUTTON "Aperte se você for o Happy", IDC_BUTTON1,
              45, 66, 48, 12, WS_TABSTOP
```

Ambas as definições são aceitáveis; no entanto, como visto em muitos exemplos anteriores, você deve adotar um estilo específico e usar somente ele no arquivo de recurso.

Você normalmente definirá os valores de identificação para cada controle dentro de um arquivo de cabeçalho separado. Para os controles com o parâmetro opcional *estilo*, as escolhas incluem os estilos *WS_TABSTOP* e *WS_GROUP*. Os estilos *WS_TABSTOP* e *WS_GROUP* controlam a interface de teclado padrão, como já descrito na Dica 1321. Você deve usar o operador *OU* bit a bit (|) para combinar todos os estilos que atribuir a um *controle*.

1326 USANDO A MACRO DIALOGBOX PARA EXIBIR UMA CAIXA DE DIÁLOGO

Como você aprendeu, seus programas definirão gabaritos de caixa de diálogo dentro do arquivo de recurso. Nas dicas a seguir, você aprenderá os diferentes modos de seus programas exibirem caixas de diálogo. Dos métodos que seus programas podem usar, o mais simples é a macro *DialogBox*. Essa macro cria uma caixa de diálogo modal a partir de um recurso de gabarito de caixa de diálogo. *DialogBox* não retorna o controle para o programa chamador até que a função de callback especificada encerre a caixa de diálogo modal chamando a função *EndDialog*. (Você aprenderá mais sobre *EndDialog* na Dica 1334.) A macro *DialogBox* usa a função *DialogBoxParam*, que a Dica 1330 discute em detalhes. O protótipo para a macro *DialogBox* é mostrado aqui:

```
int DialogBox (
   HINSTANCE hCopia,  // indicativo da ocorrência do aplicativo
   LPCTSTR lpGabarito,// identifica o gabarito da caixa de diálogo
   HWND hJanMae,      // indicativo da janela proprietária
   DLGPROC lpDialogFunc // ponteiro para procedimento de caixa de diálogo
);
```

A macro *DialogBox* aceita quatro parâmetros, como detalha a Tabela 1326

Tabela 1326 Os parâmetros para a macro ***DialogBox***.

Parâmetro	Descrição
hCopia	Identifica a ocorrência (a cópia) do aplicativo cujo arquivo executável contém o gabarito da caixa de diálogo.
lpGabarito	Identifica o gabarito de caixa de diálogo. Esse parâmetro é ou o ponteiro para uma string de caracteres terminada por *NULL* que especifica o nome do gabarito da caixa de diálogo ou um valor inteiro que especifica o identificador de recurso do gabarito da caixa de diálogo. Se o parâmetro especificar um identificador de recurso, sua palavra mais significativa precisará ser zero e sua palavra menos significativa precisará conter o identificador. Você pode usar a macro *MAKEINTRESOURCE* para criar esse valor.
hJanMae	Identifica a janela proprietária da caixa de diálogo.
lpDialogFunc	Aponta para o procedimento de caixa de diálogo. A Dica 1327 explica a função de callback *DialogProc* em detalhes.

Se a chamada à *DialogBox* for bem-sucedida, a macro retornará o parâmetro *nResult*. Seu programa usará subseqüentemente o *nResult* na chamada à função *EndDialog* que fecha a caixa de diálogo. Se a chamada à *DialogBox* falhar, a macro retornará o valor -1.

A macro *DialogBox* usa a função *CreateWindowEx* para criar a caixa de diálogo. *DialogBox* então envia uma mensagem *WM_INITDIALOG* (e uma mensagem *WM_SETFONT* se o gabarito especifica o estilo *DS_SETFONT*) ao procedimento da caixa de diálogo. A função exibe a caixa de diálogo (independentemente se o ga-

barito especifica o estilo *WS_VISIBLE*), desabilita a janela proprietária e inicia seu próprio laço de mensagem para obter e despachar as mensagens para a caixa de diálogo.

Quando o procedimento de processamento de mensagem da caixa de diálogo chamar a função *EndDialog*, *DialogBox* destruirá a caixa de diálogo, finalizará o laço de mensagem, habilitará a janela proprietária (se anteriormente desabilitada) e retornará o parâmetro *nResult* à janela chamadora.

O CD-ROM que acompanha este livro inclui o programa *caixadlg.cpp*, que cria uma caixa de diálogo na qual o usuário pode inserir um valor inteiro e uma string de caracteres. Observe que o programa define como variáveis globais, os valores inteiros e string de caracteres, que seu programa pode alterar de dentro do laço de processamento da caixa de diálogo. O fragmento de código a seguir do arquivo de programa *caixadlg.cpp* inclui a função *ProcJan* que cria a caixa de diálogo:

```
case IDM_TESTAR :
   DialogBox(hCop, "DialogoTeste", hJan, (DLGPROC)ProcTesteDialogo);
         break;
```

Quando você compilar e executar o programa *caixadlg.cpp*, sua tela exibirá a saída mostrada na Figura 1326.

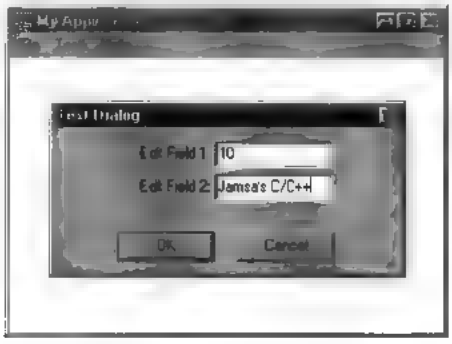

Figura 1326 A saída do programa *caixadlg.cpp*.

COMPREENDENDO O LAÇO DE MENSAGEM DA CAIXA DE DIÁLOGO

Como você aprendeu, quando seus programas criam caixas de diálogo, estas usarão seus próprios procedimentos de mensagem, em vez dos procedimentos de mensagem *ProcJan* que suas janelas-mãe usarão. Seus programas poderão ter muitos procedimentos de mensagem diferentes, até um para cada caixa de diálogo. Você pode nomear os procedimentos de mensagem da forma que quiser porque passa o endereço do procedimento cada vez que cria uma caixa de diálogo. Por exemplo, na dica anterior o programa *caixadlg.cpp* criou a caixa de diálogo com a seguinte chamada à macro *DialogBox*:

```
DialogBox(hCop, "TestaDialogo", hJan,  (DLGPROC)ProcTestaDlg);
```

O último parâmetro, o ponteiro para o procedimento de mensagem, diz ao Windows para onde ele deve enviar as mensagens destinadas à caixa de diálogo. Como você pode ver na listagem de código a seguir, o procedimento de mensagem para a caixa de diálogo executa processamento similar ao do procedimento *ProcJan*.

```
LRESULT CALLBACK ProcTestaDlg(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam)
 {
   switch(uMsg)
   {
     case WM_INITDIALOG :
          SetDlgItemInt(hDlg, IDC_EDIT1, nEditarUm, TRUE);
          SetDlgItemText(hDlg, IDC_EDIT2, szEditarDois);
          break;
     case WM_COMMAND :
          switch(LOWORD(wParam))
```

```
        {
          case IDOK :
            {
              BOOL bTran;
              nEditarUm = GetDlgItemInt(hDlg,
                                IDC_EDIT1,
                                &bTran, TRUE);
              GetDlgItemText(hDlg, IDC_EDIT2,
                                szEditarDois,
                                sizeof(szEditarDois)-1);
              EndDialog(hDlg, IDOK);
            }
            break;
          case IDCANCEL :
              EndDialog(hDlg, IDCANCEL);
              break;
        }
        break;

    default :
        return(FALSE);
    }
  return (TRUE);
  }
```

No caso do procedimento de mensagem da caixa de diálogo detalhado no fragmento de código anterior, o procedimento checa duas mensagens básicas. *WM_INITDIALOG* e *WM_COMMAND*. Se o procedimento receber a mensagem *WM_INITDIALOG*, ele inicializará os valores que os controles na caixa de diálogo exibirão. Por outro lado, se receber a mensagem *WM_COMMAND*, ele checará a palavra menos significativa do valor *wParam*. Como você sabe, a palavra menos significativa do valor *wParam* contém a constante para o comando que o usuário selecionou Se o usuário der um clique com o mouse em OK (e *wParam* contiver a constante *IDOK*), o procedimento salvará os valores dentro das caixas de edição em duas variáveis globais. Por outro lado, se o usuário der um clique com o mouse em Cancelar (e *wParam* contiver a constante *IDCANCEL*), o procedimento fechará a caixa de diálogo sem salvar quaisquer alterações que o usuário possa ter feito.

1328 Mais sobre a Manipulação de Controles

Na dica anterior, você aprendeu sobre o procedimento de mensagem da caixa de diálogo, que processa a mensagem que o Windows despacha para a caixa de diálogo. No caso do programa *caixadlg.cpp*, o procedimento de mensagem inicializa os controles e, dependendo da entrada do usuário, salva o valor de cada controle em uma variável global. O procedimento de mensagem usa as funções *SetDlgItemInt* e *SetDlgItemText* para inicializar os controles da janela e as funções *GetDlgItemInt* e *GetDlgItemText* para recuperar os valores. Você freqüentemente usará essas funções dentro de seus programas para inicializar os controles e para recuperar os valores dos controles. Os protótipos para essas quatro funções são mostrados aqui:

```
BOOL SetDlgItemInt(HWN hDlg, int nIDDlgItem, UINT uValor, BOOL bSinaliz);
BOOL SetDlgItemText(HWND hDlg, int nIDDlgItem, LPCTSTR lpString);
UINT GetDlgItemInt(HWND hDlg,-int nIDDlgItem, BOOL *lpTraduzido, BOOL
bSinaliz);
UINT GetDlgItemText(HWND hDlg, int nIDDlgItem, LPTSTR lpString, int nMaxConta);
```

Observe que ambas as funções *Set* retornam um valor *BOOL*, que indica o sucesso ou falha da função. Ambas as funções *Get* retornam um valor *UINT* (inteiro sem sinal). Para a função *GetDlgItemText*, o valor de retorno indica o número de caracteres que a função copiou para o buffer *lpString*. A Tabela 1328 lista os parâmetros para as quatro funções e mostra para quais funções cada parâmetro se aplica.

Tabela 1328 Parâmetros para as funções *SetDlgItemInt*, *SetDlgItemText*, *GetDlgItemInt* e *GetDlgItemText*.

Parâmetro	Funções	Descrição
hDlg	*SetDlgItemInt* *SetDlgItemText* *GetDlgItemInt* *GetDlgItemText*	Identifica a caixa de diálogo que contém o controle.
nIDDlgItem	*SetDlgItemInt* *SetDlgItemText*	Especifica qual controle alterar.
	GetDlgItemInt *GetDlgItemText*	Especifica o controle a partir do qual obter o valor.
uValor	*SetDlgItemInt*	Especifica o valor inteiro usado para gerar o item de texto.
bSinaliz	*SetDlgItemInt* *GetDlgItemInt*	Especifica se o parâmetro *uValor* é com ou sem sinal. Para *GetDlgItemInt*, especifica se o valor de retorno é com ou sem sinal. Se esse parâmetro for *True*, *uValor* será com sinal. Se esse parâmetro for *False* e *uValor* for menor que zero, *SetDlgItemInt* colocará um sinal de subtração antes do primeiro dígito na string. Se esse parâmetro for *False*, *uValor* será sem sinal.
lpString	*SetDlgItemText*	Para *SetDlgItemText*, *lpString* especifica a string *GetDlgItemText* a colocar dentro do controle. Para *GetDlgItemText*, *lpString* especifica o buffer de string no qual a função deve colocar o valor de retorno do controle.
lpTraduzido	*GetDlgItemInt*	Aponta para uma variável Booleana que recebe o valor sucesso/falha de uma função. *True* indica sucesso, e *False* indica falha. Esse parâmetro é opcional porque pode ser *NULL*. Quando for *NULL*, a função não retornará informações sobre sucesso ou falha.
nMaxConta	*GetDlgItemText*	Especifica o limite no número de caracteres que a função lerá do controle no buffer de string *lpString*.

COMPREENDENDO A MACRO CREATEDIALOG 1329

Na Dica 1326, você usou a macro *DialogBox* para criar uma caixa de diálogo modal a partir de um gabarito de recurso de caixa de diálogo. Como você aprendeu, seus programas freqüentemente criarão caixas de diálogo não-modais juntamente com caixas modais. Você pode usar a macro *CreateDialog* para criar caixas de diálogo não-modais. A macro *CreateDialog* cria uma caixa de diálogo não-modal a partir de um recurso de gabarito de caixa de diálogo. A macro *CreateDialog* usa a função *CreateDialogParam*, que a dica a seguir explicará em detalhes. Você implementará a macro *CreateDialog* dentro de seus programas de acordo com o formato generalizado mostrado aqui:

```
HWND  CreateDialog(
   HINSTANCE  hCopia,    // indicativo da ocorrência do aplicativo
   LPCTSTR lpNomeGabar, // identifica o gabarito da caixa de diálogo
   HWND hJanMae,        // indicativo da janela proprietária
   DLGPROC  lpFunDialogo, // ponteiro para procedimento da caixa de diálogo
);
```

Como você pode ver, a macro *CreateDialog* aceita quatro parâmetros. A Tabela 1326 já explicou os parâmetros da macro *CreateDialog* em detalhes.

A macro *CreateDialog* usa a função *CreateWindowEx* para criar a caixa de diálogo. *CreateDialog* então envia uma mensagem *WM_INITDIALOG* (e uma mensagem *WM_SETFONT* se o gabarito especifica o estilo *DS_SETFONT*) para o procedimento de caixa de diálogo. A função exibirá a caixa de diálogo se o gabarito especificar o estilo *WS_VISIBLE*. Finalmente, *CreateDialog* retorna o indicativo de janela para a caixa de diálogo.

Após *CreateDialog* retornar de seu processamento, o aplicativo usa a função *ShowWindow* para exibir a caixa de diálogo (se ela ainda não estiver visível). O aplicativo usa a função *DestroyWindow* para destruir a caixa de diálogo. Para compreender melhor o processamento que a macro *CreateDialog* executa, considere o programa *CriaDlg*, contido no CD-ROM que acompanha este livro. Quando o usuário selecionar a opção *Testar!*, a função *ProcJan* criará uma caixa de diálogo simples, como mostrado no fragmento de código a seguir:

```
case IDM_TESTAR:
       if (!hDlgNaoModal)
           hDlgNaoModal = CreateDialog(nCop, "DialogoTeste", hJan,
                       (DLGPROC)ProcDlgTeste);
       break;
```

Quando o usuário seleciona a opção *Testar!*, o programa *CriaDlg* checa se ele está exibindo atualmente a caixa de diálogo não-modal. Se estiver, ele não efetuará nenhum processamento. Caso contrário, *CriaDlg* usará a macro *CreateDialog* para exibir a caixa de diálogo não-modal.

1330 COMPREENDENDO A FUNÇÃO CREATEDIALOGPARAM

Na Dica 1320, você aprendeu como usar a macro *DialogBox* para criar caixas de diálogo modais. Na dica anterior, você aprendeu como seus programas podem usar a macro *CreateDialog* para criar caixas de diálogo não-modais. Quando seus programas precisarem criar uma caixa de diálogo não-modal, eles também poderão usar a função *CreateDialogParam*. A função *CreateDialogoParam* (que a macro *CreateDialog* chama como parte de seu processamento) cria uma caixa de diálogo não-modal a partir de um recurso de gabarito de caixa de dialogo. A função *CreateDialogParam* também permite que seus programas passem um valor definido pelo aplicativo para o procedimento de caixa de diálogo, como o parâmetro *lParam* da mensagem *WM_INITDIALOG*. Um aplicativo pode usar o valor *lParam* para inicializar os controles de caixa de diálogo. Você implementará a função *CreateDialogParam* dentro de seus programas de acordo com a forma generalizada da função, como mostrado aqui:

```
HWND CreateDialogParam(
   HINSTANCE hCopia,          // indicativo para a ocorrência do aplicativo
   LPCTSTR lpNomeGabarito,    // identifica o gabarito da caixa de diálogo
   HWND hJanMae,              // indicativo para a janela proprietária
   DLGPROC lpDialogoFunc,     // ponteiro para o procedimento da caixa de diálogo
   LPARAM dwInicParam         // valor de inicialização
);
```

A função *CreateDialogParam* espera cinco parâmetros, similar àqueles que a Tabela 1326 já definiu, com exceção do último parâmetro. O parâmetro *dwInicParam* especifica o valor a passar para o procedimento da caixa de diálogo no parâmetro *lParam* da mensagem *WM_INITDIALOG*.

A função *CreateDialogParam* usa a função *CreateWindowEx* para criar a caixa de diálogo. *CreateDialogParam* então envia uma mensagem *WM_INITDIALOG* (e uma mensagem *WM_SETFONT* se o gabarito especificar o estilo *DS_SETFONT*) para o procedimento da caixa de dialogo. A função exibirá a caixa de diálogo se o gabarito especificar o estilo *WS_VISIBLE*. Finalmente *CreateDialogParam* retornará o indicativo da janela da caixa de diálogo se teve sucesso na criação da caixa de diálogo.

Após *CreateDialogParam* retornar, seu aplicativo usará a função *ShowWindow* para exibir a caixa de diálogo (se ela ainda não estiver visível). O aplicativo usará a função *DestroyWindow* para destruir a caixa de diálogo. Para compreender melhor o processamento que *CreateDialogParam* executa, considere o programa *CreatePDialog.cpp* contido no CD-ROM que acompanha este livro. O fragmento de código a seguir da função *ProcJan* usa a função *CreateDialogParam*:

```
case IDM_TEST :
        if (!hDlgSemModo)
          {
             hDlgSemModo = CreateDialogParam(hCop,"TestDialogo",hJan,
                         (DLGPROC)TestDlgProc,
                         (LPARAM)lpMem);
    }
        break;
```

O programa *CriaPDlg.cpp* define uma estrutura do tipo *DLGDATA* para conter o status atual de cada botão na caixa de diálogo, como mostrado aqui:

```
typedef struct {
   BOOL bAssinalado;
   BOOL bRadio1;
} DLGDATA;
```

Quando o programa é inicializado (isto é, quando o procedimento *ProcJan* recebe a mensagem *WM_CREATE*), ele aloca um indicativo para uma ocorrência da estrutura. Quando o usuário seleciona *Testar!*, o programa passa o indicativo como o último parâmetro no programa *CriaPDlg.cpp*. O procedimento de diálogo *ProcDlgTesta* recebe o indicativo como o parâmetro *lParam*, e usa os valores para inicializar o estado dos controles no diálogo.

PROCESSAMENTO DE MENSAGEM PADRÃO DENTRO DE UMA CAIXA DE DIÁLOGO 1331

Você aprendeu que seus programas precisam criar uma função de processamento de mensagem de caixa de diálogo para tratar as mensagens que o Windows despacha para suas caixas de diálogo. Você também aprendeu que a função de processamento de mensagem da caixa de diálogo efetua passos similares aos da função *ProcJan*, que processa as mensagens da janela dentro de seu programa. Como você aprendeu com a função *ProcJan*, seus programas sempre devem definir a função de processamento de mensagem padrão do Windows (*DefWindowProc*) como o último *case* do seu comando *switch*, como mostrado aqui:

```
default :
        return(DefWindowProc(hJan, uMsg, wParam, lParam));
  }
```

Se você definir uma classe de janela separada para criar a janela da caixa de diálogo, deverá também definir uma função de processamento de mensagem padrão para suas caixas de diálogo, exatamente como faz dentro de sua janela-base. Para definir uma função de processamento de mensagem padrão para suas caixas de diálogo, seu programa usará a função *DefDlgProc*. A função *DefDlgProc* executa processamento de mensagem padrão para um procedimento de janela que pertence a uma classe de caixa de diálogo definida pelo aplicativo. Você implementará a função *DefDlgProc* dentro de seus programas de acordo com a seguinte forma generalizada:

```
LRESULT  DefDlgProc(
   HWND  hDlg,          // indicativo da caixa de diálogo
   UINT  Msg,           // mensagem
   WPARAM wParam,       // primeiro parâmetro da mensagem
   LPARAM lParam        // segundo parâmetro da mensagem
);
```

A função *DefDlgProc* define o procedimento de janela padrão para a classe caixa de diálogo predefinida. Esse procedimento fornece processamento interno para a caixa de diálogo encaminhando mensagens para o procedimento de caixa de diálogo e executando o processamento padrão para quaisquer mensagens que o procedimento de caixa de diálogo retorna como *False*. Os aplicativos que criam procedimentos de janela personalizados para suas caixas de diálogo personalizadas freqüentemente usam a função *DefDlgProc* em vez da função *DefWindowProc* para executar processamento de mensagem padrão.

Os aplicativos criam classes de caixa de diálogo personalizada preenchendo a estrutura *WNDCLASS* com informações apropriadas e registrando a classe com a função *RegisterClass*. Alguns aplicativos usam a função *GetClassInfo* para preencher a estrutura e especificar o nome da caixa de diálogo predefinida. Em tais casos, os aplicativos modificam pelo menos o membro *lpszClassName* antes de registrar a classe. Em todos os casos, você precisa definir o membro *cbWndExtra* de *WNDCLASS* para uma classe de caixa de diálogo personalizada para pelo menos *DLGWINDOWEXTRA*.

Um procedimento de caixa de diálogo nao deve chamar a função *DefDlgProc*; fazer isso resultará em execução recursiva. Para compreender melhor o processamento que a função *DefDlgProc* executa, considere o programa *DefDlgP.cpp*, contido no CD-ROM que acompanha este livro.

Quando você considera a definição para a função *ProcJan*, deve observar que o *switch WM_CREATE* inicializa a classe da caixa de diálogo. As definições do recurso *BlueDlg* tornam a caixa de diálogo uma classe separada, como mostrado aqui.

```
TESTDIALOG DIALOG DISCARDABLE  0, 0, 180, 70
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Diálogo de Teste"
FONT 8, "MS Sans Serif"
CLASS "BlueDlg"
BEGIN
    CHECKBOX        "Controle de caixa de seleção. ",IDC_CHECKBOX, 9,7,70,10
    GROUPBOX        "Botões de opção", -1,7,21,86,39
    RADIOBUTTON     "Primeiro", IDC_RADIO1,13,32,37,10,WS_GROUP | WS_TABSTOP
    RADIOBUTTON     "Segundo", IDC_RADIO2,13,45,39,10
    PUSHBUTTON          "Pronto", IDCANCEL,116,8,50,14,WS_GROUP
END
```

Quando você executar o programa *DefDlgP* e selecionar a opção *Testar!*, o programa usará a classe *BlueDlg* para criar a caixa de diálogo. Para quaisquer mensagens do Windows que a caixa de diálogo não processar especificamente, *DefDlgP* chamará a função de processamento de diálogo padrão (no caso do programa *DefDlgP*, a função de processamento de diálogo padrão será a função *ProcDlgTesta*).

1332 Usando a Função DlgDirList para Criar uma Caixa de Lista de Diálogo

Em dicas anteriores, você usou a macro *DialogBox*, a função *CreateDialogParam* e a função *CreateDialog* para criar caixas de diálogo simples que contêm um ou mais controles. Frequentemente, seus programas usarão caixas de diálogo para exibir informações sobre os arquivos que um disco ou uma unidade contém. Para simplificar o processo de criar caixas de diálogo que contenham informações sobre os arquivos em um disco ou unidade, a API Win32 fornece a função *DlgDirList*. Seus programas usarão a função *DlgDirList* para preencher uma caixa de lista especificada com os nomes de todos os arquivos que atendam ao caminho especificado ou ao nome de arquivo. Você implementará a função *DlgDirList* como mostrado no protótipo a seguir:

```
int DlgDirList(
   HWND  hDlg,          // indicativo para a caixa de diálogo com caixa de
                           lista
   LPTSTR lpCamEspec,   // ponteiro para a string do caminho ou do nome de
                           arquivo
   int nIDCxLista,      // identificador da caixa de lista
   int nIDCamEstatico,  // identificador de controle estático
   UINT uTipoArq        // atributos de arquivo a exibir
);
```

Como você pode ver, a função *DlgDirList* aceita cinco parâmetros. A Tabela 1332.1 explica esses parâmetros em detalhes.

Tabela 1332.1 Os parâmetros para a função ***DlgDirList***.

Parâmetros	Descrição
hDlg	Identifica a caixa de diálogo que contém a caixa de lista.
lpCamEspec	Aponta para uma string terminada por *NULL* que contém o caminho ou o nome de arquivo. *DlgDirList* modifica essa string, que deve ser grande o suficiente para conter as modificações.
nIDCxLista	Especifica o identificador de uma caixa de lista. Se esse parâmetro for zero, *DlgDirList* assumirá que nenhuma caixa de lista existe e não tentará preencher uma.

Tabela 1332.1 Os parâmetros para a função ***DlgDirList*** *(Continuação)*

Parâmetros	Descrição
nIDCamEstatico	Especifica o identificador do controle estático que a caixa de diálogo usa para exibir a unidade e o diretório atuais. Se esse parâmetro for zero, *DlgDirList* assumirá que esse controle não está presente.
uTipoArq	Especifica atributos dos nomes de arquivo que a caixa de lista deverá exibir. Esse parametro precisa ser um ou mais dos valores que a Tabela 1332.2 detalha.

Como a Tabela 1332.1 detalhou, o parâmetro *uTipoArq* aceita diversos valores possíveis. Quando você usar os valores possíveis para o parâmetro *uTipoArq*, usará o operador *OU* bit a bit para combinar os valores. A Tabela 1332.2 lista os valores possíveis para o parâmetro *uTipoArq*.

Tabela 1332.2 Valores possíveis para o parâmetro ***uTipoArq***.

Valor	Descrição
DDL_ARCHIVE	Inclui os arquivos para backup.
DDL_DIRECTORY	Inclui subdiretórios. Os nomes dos subdiretórios são delimitados por colchetes ([ ]).
DDL_DRIVES	Inclui as unidades. As unidades são listadas na forma [-*x*-], onde *x* é a letra da unidade
DDL_EXCLUSIVE	Inclui apenas os arquivos com os atributos especificados. Por padrão, os arquivos de leitura/gravação são listados mesmo se você não especifica *DDL_READWRITE*.
DDL_HIDDEN	Inclui os arquivos ocultos.
DDL_READONLY	Inclui os arquivos somente leitura.
DDL_READWRITE	Inclui os arquivos de leitura/gravação sem atributos adicionais.
DDL_SYSTEM	Inclui os arquivos do sistema.
DDL_POSTMSGS	Encaminha mensagens para a fila de mensagens do aplicativo Por padrão, *DlgDirList* envia mensagens diretamente para o procedimento de mensagem da caixa de diálogo.

Se a função *DlgDirList* exibir uma lista — mesmo se for uma lista vazia —, a função retornará um valor diferente de zero. Se a string de entrada não contiver um caminho de pesquisa válido (ou se um erro impedir a geração da lista), o valor de retorno será zero.

Se você especificar uma string de tamanho zero, ou somente um nome de diretório sem nome de arquivo para o parâmetro *lpCamEspec*, a função *DlgDirList* mudará a string para "*.*". O parâmetro *lpCamEspec* tem a seguinte forma:

```
[unidade:] [[\u]diretório[\diretório]\u]  [nomearq]
```

Neste exemplo, *unidade* é a letra da unidade, *diretório* é um nome válido de diretório e *nomearq* é um nome de arquivo que precisa conter pelo menos um caractere-chave (? ou *). Se *lpCamEspec* incluir um nome de unidade ou de diretório, ou ambos, a função modificará a unidade e o diretório atuais para a unidade e o diretório especificados antes de a função preencher a caixa de lista. A função também atualizará o controle estático que o parâmetro *nIDCamEstatico* identifica com o novo nome de unidade ou de diretório, ou ambos.

Após a função preencher a caixa de lista, *DlgDirList* atualiza *lpCamEspec* removendo a porção da unidade e do diretório, ou ambos, do caminho e do nome de arquivo. Em seguida, *DlgDirList* envia as mensagens *LB_RESETCONTENT* e *LB_DIR* para a caixa de lista. Para compreender melhor o processamento que a função *DlgDirList* executa, considere o programa *list_dir.cpp*, contido no CD-ROM que acompanha este livro. O arquivo *list_dir.cpp* define a caixa de diálogo com a caixa de texto estático e o botão de comando fecha o diálogo, como mostrado aqui:

```
TESTDIALOG DIALOG DISCARDABLE  20, 20, 150, 110
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Diálogo de Teste"
```

```
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_DIRECTORY,6,5,136,13, ES_AUTOHSCROLL | ES_READONLY
    | NOT WS_TABSTOP
    LISTBOX     IDC_LIST,6,20,136,59,LBS_SORT | LBS_NOINTEGRALHEIGHT |
    LBS_DISABLENOSCROLL | WS_VSCROLL | WS_TABSTOP
    PUSHBUTTON      "Done", IDCANCEL,50,87,50,14, WS_GROUP
END
```

O arquivo *list_dir.cpp* usa a função *DlgDirList* para criar a caixa de diálogo a partir de dentro da função *ProcJan*. O programa cria a caixa de diálogo quando o usuário seleciona a função *Testar!*. A caixa de diálogo contém uma caixa de lista que exibe o conteúdo do diretório atual e uma caixa de texto estático que exibe o nome do diretório. Se o usuário der um clique duplo com o mouse em qualquer arquivo dentro da caixa de lista, o programa mudará para aquele diretório ou unidade se a seleção for um diretório ou unidade. O programa então exibirá a seleção dentro da caixa de texto estática, independentemente do tipo da seleção. Quando você compilar e executar o programa *list_dir.cpp*, sua tela exibirá a saída mostrada na Figura 1332.

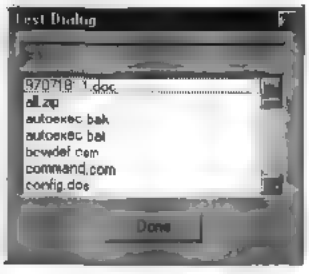

Figura 1332 A saída do programa *list_dir.cpp*.

1333 RESPONDENDO ÀS SELEÇÕES DO USUÁRIO DENTRO DA CAIXA DE LISTA

Na dica anterior, você criou o programa *list_dir.cpp*, que exibe os arquivos dentro do diretório atual quando o programa exibe uma caixa de diálogo. Como você aprendeu, a função *ProcDlgTesta* do programa mudou as informações dentro do texto estático quando o usuário deu um clique duplo com o mouse em um item dentro da caixa de lista.

Em dicas posteriores, você aprenderá sobre a captura da entrada do usuário em detalhes. No entanto, dentro do programa *list_dir.cpp*, o processamento é relativamente simples: primeiro o programa verifica se o usuário deu um clique duplo no botão do mouse. Se deu, o programa verifica a função *DlgDirSelectEx* para determinar o que o usuário selecionou — um diretório ou um arquivo. A função *DlgDirSelectEx* recupera a seleção atual de uma caixa de lista de seleção única. A função assume que a função *DlgDirList* preencheu a caixa de lista e que a seleção é uma letra de unidade, nome de arquivo ou nome de diretório. Seus programas implementarão a função *DlgDirSelectEx* como mostrado aqui:

```
BOOL DlgDirSelectEx(
    HWND hDlg,          // indicativo para a caixa de diálogo com caixa de lista
    LPTSTR lpString,    // ponteiro para buffer para string do caminho
    int nConta,         // número de caracteres na string do caminho
    int nIDCxLista      // identificador da caixa de lista
);
```

Se a seleção atual for o nome de um diretório, a função retornará um valor diferente de zero. Se a seleção atual não for um nome de diretório, a função retornará zero. A função *DlgDirSelectEx* copia a seleção para o buffer para o qual o parâmetro *lpString* aponta. Se a seleção atual for um nome de diretório ou uma letra de unidade, *DlgDirSelectEx* removerá os colchetes delimitadores (e os hífens, para as letras de unidades), para que o nome ou a letra esteja pronto para a função *DlgDirList* inserir o nome ou a letra em um novo caminho. Se não houver uma seleção, *lpString* não mudará.

A função *DlgDirSelectEx* envia *LB_GETCURSEL* e *LB_GETTEXT* para a caixa de lista. A função não permite que a caixa de lista retorne mais de um nome de arquivo. A caixa de lista não pode ser uma caixa de lista de múltiplas seleções. Se for uma caixa de lista de múltiplas seleções, *DlgDirSelectEx* não retornará um valor zero, e *lpString* permanecerá inalterada. Quando você chamar a função *DlgDirSelectEx*, seu programa deverá testar o valor de retorno e responder apropriadamente, como a função *TestDlgProc* do programa *List_Dir* mostra:

```
LRESULT CALLBACK TestDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM
lParam)
 {
   static char szTmp[255];

   switch(uMsg);
   {
      case WM_INITDIALOG :
         DlgDirList(hDlg, "*.*", IDC_LIST, IDC_DIRECTORY,
                    DDL_DIRECTORY | DDL_DRIVES);
         break;
      case WM_COMMAND :
         switch(LOWORD(wParam))
         {
           case IDC_LIST :
              if (HIWORD(wParam) == LBN_DBLCLK)
              {
                if (DlgDirSelectEx(hDlg, szTmp, sizeof(szTmp), IDC_LIST))
                {
                   strcat(szTmp, "*.*");
                   DlgDirList(hDlg, szTmp, IDC_LIST, IDC_DIRECTORY,
                   DDL_DIRECTORY | DDL_DRIVES);
                }
                else
                   MessageBox(hDlg, szTmp, "Arquivo selecionado",
                              MB_OK | MB_ICONINFORMATION);
              }

              break;
            case IDCANCEL :
               EndDialog(hDlg, IDCANCEL);
               break;
         }

         break;
   default :
         return(FALSE);
   }
   return(TRUE);
 }
```

FECHANDO A CAIXA DE DIÁLOGO 1334

Nas dicas anteriores, você aprendeu como criar e exibir diversas caixas de diálogo. Como foi visto em listagens de código em dicas anteriores, você sempre precisa usar a função *EndDialog* para fechar suas caixas de diálogo modais. A função *EndDialog* destrói uma caixa de diálogo modal, o que faz o sistema terminar qualquer processamento para a caixa de diálogo. Você usará a função *EndDialog* dentro de seus programas como mostrado no protótipo a seguir:

```
BOOL EndDialog(
   HWND hDlg,      // indicativo para caixa de diálogo
   int nResult     // valor a retornar
);
```

O parâmetro *hDlg* identifica a caixa de diálogo que a função *EndDialog* deverá destruir. O parâmetro *nResult* permite que seus programas especifiquem um valor de retorno da função que criou a caixa de diálogo para o aplicativo chamador. Você precisa usar a função *EndDialog* para destruir caixas de diálogo que criar com *DialogBox*, *DialogBoxParam*, *DialogBoxIndirect* e *DialogBoxIndirectParam*. Um aplicativo chama *EndDialog* a partir de dentro do procedimento de caixa de diálogo. Não use a funçao *EndDialog* para qualquer outro propósito.

Um procedimento de caixa de diálogo pode chamar a função *EndDialog* a qualquer momento, até mesmo durante o processamento da mensagem *WM_INITDIALOG*. Se seu aplicativo chamar *EndDialog* enquanto estiver processando a mensagem *WM_INITDIALOG*, o Windows destruirá a caixa de diálogo antes que ela apareça e antes do Windows definir o foco de entrada para a caixa de diálogo.

EndDialog não destrói a caixa de diálogo imediatamente. Em vez disso, ela define um sinalizador e permite que o procedimento de caixa de diálogo retorne o controle para o sistema. O sistema verifica o sinalizador antes de tentar recuperar a próxima mensagem da fila do aplicativo. Se *EndDialog* tiver definido anteriormente o sinalizador, o sistema finalizará o laço de mensagem, destruirá a caixa de diálogo e usará o valor em *nResult* como o valor de retorno da função que criou a caixa de diálogo.

1335 COMPREENDENDO A ENTRADA DO USUÁRIO

Em dicas anteriores, você criou vários tipos diferentes de programas Windows. No entanto, todos os programas tiveram uma característica em comum: recebiam digitações do usuário para efetuar seu processamento — mesmo se a digitação fosse simplesmente um comando *Fechar*. Como você aprendeu, a vasta maioria dos programas Windows que você desenvolver (com exceção dos servidores automatizados) requer intervenção do usuário de um tipo ou de outro para efetuar processamento útil. Os programas de servidor automatizados são uma exceção notável porque sua entrada vem totalmente de outros programas e não dos usuários.

No Windows, um programa recebe entrada de dados do usuário a partir de qualquer um dentre vários dispositivos. No entanto, os dispositivos mais comumente usados são o mouse e o teclado. Outros dispositivos de entrada que o usuário pode usar incluem uma caneta de luz, uma tela de toque, um *joystick* e assim por diante. Como você aprendeu anteriormente, toda vez que o usuário digita uma tecla no teclado (e toda vez que o usuário move ou dá um clique no mouse), o Windows gera uma mensagem. Essa mensagem pode ir para o sistema operacional, para a janela do seu programa, ou para uma caixa de diálogo. Em alguns casos, essa mensagem pode ir para múltiplas localizações.

No Windows, seus programas geralmente usarão controles predefinidos, tais como botões, controles de edição e menus para responder ao teclado e ao mouse de forma apropriada. Nas dicas a seguir, você aprenderá sobre os métodos adicionais que seus programas podem usar para controlar melhor a resposta à entrada de dados feita pelo teclado ou pelo mouse.

1336 RESPONDENDO AOS EVENTOS DO MOUSE

Como você aprendeu, seus programas responderão à entrada do usuário na forma de mensagens do sistema. Em dicas anteriores, seus programas verificaram a mensagem *WM_COMMAND* para determinar se o usuário enviou um comando para o seu programa e se deve ou não responder à seleção de menu ou de controle que gerou o comando. Quando seus programas trabalham com atividades do mouse que o usuário executa, seus programas precisam verificar as mensagens listadas na Tabela 1336.

Tabela 1336 Mensagens de mouse do Windows.

Mensagens	Significado
WM_CAPTURECHANGED	O Windows envia a mensagem *WM_CAPTURECHANGED* para a janela que está perdendo a captura do mouse.
WM_LBUTTONDBLCLK	O usuário pressionou duas vezes o botão esquerdo do mouse.

Tabela 1336 Mensagens de mouse do Windows. (*Continuação*)

Mensagens	Significado
WM_LBUTTONDOWN	O usuário pressionou o botão esquerdo do mouse, e ele ainda está pressionado.
WM_LBUTTONUP	O usuário soltou o botão esquerdo do mouse.
WM MBUTTONDBLCLK	O usuário pressionou o botão central do mouse duas vezes (somente mouse de três botões).
WM_MBUTTONDOWN	O usuário pressionou o botão central do mouse e ele ainda está pressionado (somente mouse de três botões).
WM_MBUTTONUP	O usuário soltou o botão central do mouse (somente mouse de três botões).
WM_MOUSEACTIVATE	O usuário deu um clique com o mouse dentro de uma janela atualmente inativa.
WM_MOUSEMOVE	O usuário moveu o mouse dentro da janela.
WM_NCLBUTTONDBLCLK	O usuário pressionou o botão esquerdo do mouse duas vezes dentro de uma área não-cliente da janela e ele ainda está pressionado.
WM_NCLBUTTONDOWN	O usuário presionou o botão esquerdo do mouse dentro de uma área não-cliente da janela e ele ainda está pressionado.
WM_NCLBUTTONUP	O usuário soltou o botão esquerdo do mouse dentro de uma área não-cliente da janela.
WM_NCMBUTTONDBLCLK	O usuário pressionou o botão central do mouse duas vezes dentro da área não-cliente da janela, e ainda está pressionado (somente mouse de três botões somente).
WM_NCMBUTTONDOWN	O usuário pressionou o botão central do mouse dentro da área não-cliente de uma janela, e ele ainda está pressionado (somente mouse de três botões)
WM_NCMBUTTONUP	O usuário soltou o botão central do mouse dentro da área não-cliente de uma janela (somente mouse de três botões).
WM_NCMMOUSEMOVE	O usuário moveu o mouse dentro da área não-cliente de uma janela.
WM_NCRBUTTONDBLCLK	O usuário pressionou duas vezes o botão direito do mouse dentro de uma área não-cliente da janela.
WM NCRBUTTONDOWN	O usuário pressionou o botão direito do mouse dentro de uma área não-cliente de uma janela, e ainda está pressionado.
WM_NCRBUTTONUP	O usuário soltou o botão direito do mouse, dentro da área não-cliente de uma janela.
WM_RBUTTONDBLCLK	O usuário pressionou duas vezes o botão direito do mouse.
WM RBUTTONDOWN	O usuário pressionou o botão direito do mouse e ele ainda está pressionado.
WM_RBUTTONUP	O usuário soltou o botão direito do mouse.

Quando um aplicativo recebe uma mensagem do mouse, o valor *lParam* contém as posições X e Y do cursor na tela. O Windows armazena a posição Y na palavra de ordem mais significativa de lParam e a posição X na palavra menos significativa. Seu programa deve usar as macros *LOWORD* e *HIWORD* para extrair os dois valores. A dica a seguir examina a mensagem *WM_MOUSEMOVE* como um exemplo de uma mensagem de mouse do Windows e inclui um fragmento de código que mostra como você extrai os valores do parâmetro *lParam*.

USANDO A MENSAGEM WM_MOUSEMOVE 1337

Vimos na dica anterior que, o Windows gera mensagens para os seus aplicativos sempre que o usuario manipula o mouse. Uma das mensagens mais comuns que o Windows enviará para os seus programas é *WM_MOUSEMOVE*. O Windows encaminhará a mensagem *WM_MOUSEMOVE* para uma janela toda vez que o cursor for

movido. Se uma janela não tiver capturado o mouse antes, o Windows encaminhará a mensagem *WM_MOUSEMOVE* para a janela que contém o cursor. Caso contrário, o Windows encaminhará a mensagem *WM_MOUSEMOVE* para a janela que capturou o mouse. Quando seu programa receber uma mensagem *WM_MOUSEMOVE*, deverá verificar os seguintes valores antes de começar seu processamento respondente:

```
fwTeclas = wParam;        // sinalizadores de tecla
xPos = LOWORD(lParam);    // posição horizontal do cursor
yPos = HIWORD(lParam);    // posição vertical do cursor
```

Como você aprendeu na dica anterior, o parâmetro *lParam* contém as posições X e Y do cursor. O parâmetro *wParam* conterá os valores *fwTeclas*. O valor *fwTeclas* indica se várias teclas virtuais estão pressionadas. O valor *fwTeclas* pode ser qualquer combinação dos valores mostrados na Tabela 1337.

Tabela 1337 Valores possíveis para o valor *fwTeclas*.

Valor	Descrição
MK_CONTROL	Ligado se a tecla Ctrl estiver pressionada.
MK_LBUTTON	Ligado se o botão esquerdo do mouse estiver pressionado.
MK_MBUTTON	Ligado se o botão central do mouse estiver pressionado.
MK_RBUTTON	Ligado se o botão direito do mouse estiver pressionado.
MK_SHIFT	Ligado se a tecla Shift estiver pressionada.

Para capturar os movimentos do mouse a partir de dentro de seus procedimentos de mensagem, seus programas devem usar código similar ao que está no fragmento a seguir:

```
case WM_MOUSEMOVE :
    {
       nXPos = LOWORD(lParam);
       nYPos = HIWORD(lParam);
       // outros comandos
```

No fragmento precedente, o programa atribui a posição atual do mouse aos valores *nXPos* e *nYPos* toda vez que o usuário move o mouse.

1338 LENDO OS BOTÕES DO MOUSE

Na dica anterior, você aprendeu como usar a mensagem de janela *WM_MOUSEMOVE* para que seus programas respondam aos movimentos do mouse na tela. Similarmente, acrescentar código aos seus programas para fazê-los responder às atividades de botão do mouse também é simples. Por exemplo, se você quiser que seus programas respondam quando o usuário der um clique duplo no mouse em qualquer lugar dentro da área cliente da janela, seus programas usarão código dentro da função *ProcJan* similar ao seguinte:

```
LRESULT CALLBACK ProcJan(HWND hJan, UINT uMsg, WPARAM wParam, LPARAM lParam)
  {
    switch(uMsg)
    {
       case WM_LBUTTONDBLCLK :
            MessageBox(hJan, "Clique duplo no botão", NULL, MB_OK);
            break;
```

Se, por outro lado, você quiser que seu programa responda de forma diferente se o usuário mantiver a tecla Ctrl pressionada enquanto pressiona o mouse, você poderá construir o fragmento de código como mostrado aqui:

```
case WM_LBUTTONDBLCLK :
     if ((wParam && MK_CONTROL) == MK_CONTROL)
     {
        MessageBox(hJan, "Clique duplo no botão com a tecla Ctrl
pressionada", NULL, MB_OK);
```

```
        break;
    }
    else
    {
        MessageBox(hJan, "Clique duplo no botão sem que a tecla CTRL esteja
pressionada", NULL, MB_OK);
        break;
    }
```

Como você pode ver a partir do código, seus programas podem verificar se o usuário está atualmente pressionando uma determinada tecla virtual efetuando uma operação *E* bit a bit no valor *wParam* e o valor da tecla virtual que você quer conferir. À medida que seus programas se tornarem mais complexos, você freqüentemente testará as teclas virtuais ao processar os eventos do mouse. Embora o Windows passe somente certas teclas virtuais dentro do parâmetro *wParam* em conjunção com um clique do mouse, existe um grande número de teclas virtuais que o Windows suporta, como você aprenderá na Dica 1340, logo à frente.

RESPONDENDO AOS EVENTOS DO TECLADO 1339

Como você aprendeu, o Windows despachará mensagens para seus aplicativos toda vez que o usuário efetuar alguma atividade com o mouse. O Windows também despachará mensagens para seus aplicativos toda vez que o usuário digitar uma tecla no teclado do computador. A mensagem que o usuário despachará será uma das listadas na Tabela 1339.

Tabela 1339 As mensagens do sistema que o Windows envia para o processamento das teclas.

Mensagem	Descrição
WM_ACTIVATE	A tecla está dentro de uma janela inativa.
WM_CHAR	O código ASCII para a letra se o usuário pressionou um caractere no teclado.
WM_GETHOTKEY	Permite que seus programas recuperem a tecla de ativação anteriormente definida para uma janela.
WM_HOTKEY	O usuário pressionou uma tecla de ativação.
WM_KEYDOWN	O usuário pressionou uma tecla.
WM_KEYUP	O usuário soltou a tecla pressionada.
WM_KILLFOCUS	Enviada para uma janela imediatamente após a janela perder o foco do teclado.
WM_SETFOCUS	Enviada para uma janela imediatamente após a janela receber o foco do teclado.
WM_SETHOTKEY	Permite que seus programas definam uma "tecla de ativação" para uma janela.
WM_SYSCHAR	O código ASCII para a letra que o usuário soltou enquanto simultaneamente soltava a tecla Alt.
WM_SYSKEYDOWN	O usuário soltou uma tecla enquanto pressionava simultaneamente a tecla Alt.
WM_SYSKEYUP	O usuário soltou a tecla pressionada e a tecla Alt pressionada.

A função *TranslateMessage* no laço de mensagem do encadeamento gerará a mensagem *WM_CHAR* se reconhecer o caractere como um caractere ASCII. Geralmente, seus aplicativos usarão a mensagem *WM_KEYDOWN* para verificar as teclas de função, teclas de cursor, o teclado numérico reduzido e as teclas de edição, tais como PageUp e PageDn. Essas teclas fazem o melhor uso dos códigos de tecla virtual, sobre os quais você aprenderá mais na dica a seguir.

Por outro lado, seus programas devem usar *WM_CHAR* para recuperar letras, números e símbolos imprimíveis. Usando a mensagem *WM_CHAR* para processar essas digitações é melhor porque ASCII atribui valores diferentes para as letras minúsculas e maiúsculas. Com *WM_KEYDOWN*, seus aplicativos precisam verificar o caractere que o usuário digitou, bem como o estado atual da tecla Shift. A mensagem *WM_CHAR* trata o processamento da tecla Shift para você.

Se o usuário pressiona a tecla Alt enquanto pressiona outra tecla, o aplicativo receberá a seguinte seqüência de mensagem: *WM_SYSKEYDOWN, WM_SYSCHAR, WM_SYSKEYUP*. Seus programas devem usar essas mensagens para verificar seqüências específicas da tecla Alt.

1340 COMPREENDENDO AS TECLAS VIRTUAIS

Vimos na dica anterior que, caso seus programas possam processar uma tecla como um caractere ASCII, elas devem usar a mensagem *WM_CHAR* para fazer isso. No entanto, muitas vezes seus programas podem requerer ou permitir que o usuário use outras teclas além das teclas ASCII normais — por exemplo, as teclas de setas, as teclas numéricas no teclado numérico reduzido e assim por diante. Quando você controla essas teclas, deve usar os valores de teclas virtuais para fazer isso. As teclas virtuais o liberam de considerar qual tipo de teclado o usuário terá, porque o código de tecla virtual para a primeira tecla de função sempre deve ser o mesmo, independente do fabricante ou modelo do teclado. As mensagens *WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN* e *WM_SYSKEYUP* enviam os códigos de tecla virtual como o valor *wParam* da mensagem.

Dentro da estrutura de código de tecla virtual, os números do teclado numérico reduzido têm seus próprios códigos de tecla virtual. Adicionalmente, o código virtual para as teclas de caractere e numéricas é seu equivalente ASCII maiúsculo (em outras palavras tanto a quanto A são *VK_A*). Finalmente, observe que as teclas Shift geram o mesmo código de tecla virtual. A Tabela 1340 lista os códigos de tecla virtual definidos pela API Win 32.

Tabela 1340 Os códigos de tecla virtual definidos na API Win 32.

Código de Tecla	Valor (Hexa)	Equivalente no Teclado ou Mouse
VK_LBUTTON	01	Botão esquerdo do mouse
VK_RBUTTON	02	Botão direito do mouse
VK_CANCEL	03	Processamento de Ctrl-Break
VK_MBUTTON	04	Botão central do mouse (mouse de três botões) ou ambos os botões esquerdo e direito simultaneamente
VK_BACK	08	Backspace
VK_TAB	09	Tab
VK_CLEAR	0C	Clear
VK_RETURN	0D	Enter
VK_SHIFT	10	Shift
VK_CONTROL	11	Ctrl
VK_MENU	12	Alt
VK_PAUSE	13	Pause
VK_CAPITAL	14	Caps Lock
VK_ESCAPE	18	Esc
VK_SPACE	20	Barra de espaço
VK_PRIOR	21	PageUp
VK_NEXT	22	PageDn
VK_END	23	End
VK_HOME	24	Home
VK_LEFT	25	Seta para a esquerda
VK_UP	26	Seta para cima
VK_RIGHT	27	Seta para a direita
VK_DOWN	28	Seta para baixo
VK_SELECT	29	Select
VK_EXECUTE	2B	Execute
VK_SNAPSHOT	2C	Print Screen

Tabela 1340 Os códigos de tecla virtual definidos na API Win 32. (*Continuação*)

Código de Tecla	Valor (Hexa)	Equivalente no Teclado ou Mouse
VK_INSERT	2D	Insert
VK_DELETE	2E	Delete
VK_HELP	2F	Help
VK_0-VK_9	30-39	0-9
VK_A-VK_Z	41-5A	A-Z
VK_NUMPAD0	60	0 do teclado numérico reduzido
VK_NUMPAD1	61	1 do teclado numérico reduzido
VK_NUMPAD2	62	2 do teclado numérico reduzido
VK_NUMPAD3	63	3 do teclado numérico reduzido
VK_NUMPAD4	64	4 do teclado numérico reduzido
VK_NUMPAD5	65	5 do teclado numérico reduzido
VK_NUMPAD6	66	6 do teclado numérico reduzido
VK_NUMPAD7	67	7 do teclado numérico reduzido
VK_NUMPAD8	68	8 do teclado numérico reduzido
VK_NUMPAD9	69	9 do teclado numérico reduzido
VK_MULTIPLY	6A	Tecla de multiplicação
VK_ADD	6B	Tecla da adição
VK_SEPARATOR	6C	Tecla do separador
VK_SUBTRACT	6D	Tecla da subtração
VK_DECIMAL	6E	Tecla decimal
VK_DIVIDE	6F	Tecla da divisão
VK_F1-VK_F24	70-87	Teclas de função F1-F24
VK_NUMLOCK	90	NumLock
VK_SCROLL	91	ScrollLock

USANDO AS TECLAS VIRTUAIS 1341

Como você aprendeu, seus programas geralmente receberão um dos três tipos de entrada de teclado do Windows: entrada de caractere ASCII reconhecido (dentro do parâmetro *wParam* de uma mensagem *WM_CHAR*); entrada de caractere que não é do sistema não-reconhecida dentro do *wParam* de uma mensagem *WM_KEYDOWN*; e entrada de caractere do sistema dentro do *wParam* da mensagem *WM_SYSCHAR* ou *WM_SYSKEYDOWN*. Você também aprendeu como seus programas podem facilmente processar as mensagens entrantes do mouse. Processar mensagens do teclado também é fácil. Por exemplo, o código a seguir exibirá caracteres reconhecidos dentro de uma caixa de edição de texto — se o caractere não-reconhecido for uma tecla de função, o código exibirá uma caixa de mensagem que informa isso:

```
LRESULT CALLBACK DlgTestProc(HWND hJan, UINT uMsg,
                             WPARAM wParam, LPARAM lParam)
{
  switch(uMsg)
  {
     case WM_CHAR :
        GetDlgItem(hJan, IDC_EDITBOX);
        lstrcat(szEditBox, wParam);
        SetDlgItemText(hJan, IDC_EDITBOX, szEditBox);
        break;
```

```
      case WM_KEYDOWN :
         switch(wParam)
         {
           case VK_F1 :
             MessageBox(hJan,
                "Função 1 Tecla Pressionada", NULL, MB_OK);
             break;
           case VK_F2 :
             MessageBox(hJan,
                "Função 2 Tecla Pressionada", NULL, MB_OK);
             break;
// Mais testes de tecla virtual aqui
         }
```

Como você pode ver, o programa verifica os valores de tecla da função virtual que o Windows armazena dentro dos parâmetros *wParam* para determinar qual tecla virtual o usuário pressionou.

1342 MAIS SOBRE O USO DA MENSAGEM WM_KEYDOWN

O Windows encaminha a mensagem *WM_KEYDOWN* para a janela com o foco do teclado quando o usuário pressiona uma tecla que não é do sistema. Uma tecla que não é do sistema, é uma tecla que o usuário pressiona sem pressionar simultaneamente a tecla Alt. Como você também viu na dica anterior, o Windows passa o código de tecla virtual da tecla pressionada pelo usuário dentro do parâmetro *wParam*. No entanto, o Windows também passa o valor *lKeyData* dentro do parâmetro *lParam*. O parâmetro *lKeyData* especifica o número de repetição, o código de varredura, o sinalizador de tecla estendida, o código de contexto, o sinalizador de estado da tecla anterior e o sinalizador do estado da transição, como mostrado na Tabela 1342.1

Tabela 1342.1 Os valores que o Windows passa dentro do parâmetro ***lParam*** com a mensagem ***WM_KEYDOWN***.

Valor	Descrição
0-15	Especifica o número de repetição. O valor será o número de vezes que a tecla for repetida se o usuário mantiver uma tecla pressionada.
16-23	Especifica o código de varredura. O valor depende do fabricante original do equipamento (OEM).
24	Especifica se a tecla será uma tecla estendida, tal como as teclas Alt e Ctrl da direita que aparecem em um teclado estendido de 101-ou 102 teclas. Quando o bit estiver ligado, a tecla será uma tecla estendida; caso contrário, a tecla não será uma tecla estendida.
25-28	Reservado; não use.
29	Especifica o código do contexto. O valor é sempre 0 para uma mensagem *WM_KEYDOWN*
30	Especifica o estado anterior da tecla. O bit estará ligado se a tecla estiver pressionada antes de a mensagem ser enviada, e não estará ligado se a tecla estiver liberada.
31	*Especifica o estado da transmissão. O bit nunca está ligado para uma mensagem WM_KEYDOWN.*

Se o usuário pressionar a tecla de função F10, a função *DefWindowProc* definirá um sinalizador interno. Quando *DefWindowProc* recebe uma mensagem *WM_KEYUP*, a função verifica se o sinalizador interno está ligado e, em caso afirmativo, envia uma mensagem *WM_SYSCOMMAND* para a janela de alto nível *DefWindowProc* define o parâmetro *wParam* da mensagem para *SC_KEYMENU*.

Devido ao seu recurso de auto-repetição, o Windows pode encaminhar mais de uma mensagem *WM_KEYDOWN* para seu aplicativo antes de encaminhar uma mensagem *WM_KEYUP*. Seus programas podem usar o estado anterior da tecla (bit 30) para determinar se a mensagem *WM_KEYDOWN* indica ou não a primeira transição pressionada ou uma transição repetida.

Para os teclados de 101 e de 102 teclas, as teclas estendidas são Alt e Ctrl da direita na seção principal do teclado; Ins, Del, Home, End, PageUp, PageDn e as teclas de setas no conjunto à esquerda do teclado numérico reduzido; a tecla de divisão (/) e a tecla Enter nesse teclado. Outros teclados podem suportar o bit de tecla estendida no parâmetro *lKeyData*.

Algumas vezes você irá querer que seus programas recebam uma string que represente o nome de uma tecla. Na dica anterior, você processou as teclas conhecidas dentro de um comando *switch*. No entanto, a API Win32 mantém uma lista das teclas para as quais todos os códigos virtuais correspondem. Você também pode usar a função *GetKeyNameText* para recuperar essa lista. A função *GetKeyNameText* recupera uma string que corresponde ao nome de uma tecla. Você chamará a função *GetKeyNameText* dentro de seus programas, como mostrado no protótipo a seguir:

```
int GetKeyNameText(
   LONG lParam,       // segundo parâmetro da mensagem do teclado
   LPTSTR lpString,   // endereço do buffer para nome da tecla
   int nTamanho       // comprimento máximo da string
);
```

O parâmetro *lpString* aponta para um buffer que receberá o nome-chave. O parâmetrd *nTamanho* especifica o comprimento máximo, em caracteres, do nome-chave, incluindo o caractere *NULL* finalizador. (O parâmetro *nTamanho* deve ser igual ao tamanho do buffer para o qual o parâmetro *lpString* aponta.) O parâmetro *lParam* especifica o segundo parâmetro da mensagem do teclado (tal como *WM_KEYDOWN*) que *GetKeyNameText* processará. A função interpreta as porções de *lParam* que a Tabela 1342.2 detalha.

Tabela 1342.2 Os bits dentro de ***lParam*** que ***GetKeyNameText*** interpreta.

Bits	Significado
16-23	Código de varredura
24	Sinalizador de tecla estendida. Distingue algumas teclas em um teclado estendido.
25	Bit "não importa". O aplicativo que chama a função *GetKeyNameText* define esse bit para indicar que a função não deve distingur entre as teclas Ctrl e Shift da direita e da esquerda, por exemplo.

O formato da string nome-chave depende do layout atual do teclado. O controlador de teclado mantém uma lista dos nomes na forma de strings de caractere para as teclas com nomes maiores que um único caractere. O controlador de teclado traduz o nome da tecla de acordo com o layout do teclado instalado no momento. O nome de uma tecla de caractere é o próprio caractere. O CD-ROM que acompanha este livro inclui o programa *Exib_Tec.cpp*, que exibe um nome de tecla toda vez que o usuário pressiona uma tecla.

Como você verá, dentro do case *WIN_KEYDOWN*, o programa realiza processamento significativo relacionado com dispositivos de contexto, o que você aprenderá em dicas posteriores. A chamada de função importante para o propósito desta dica é a chamada à *GetKeyNameText*, como mostrado aqui:

```
GetKeyNameText(lParam, szNome, 30);
```

O programa simplesmente chama *GetKeyNameText*, depois exibe o valor dentro do buffer *szName* na janela.

DEFININDO E RETORNANDO O TEMPO DO CLIQUE DUPLO NO MOUSE 1343

Seus programas efetuarão processamento significativo para pegar e corretamente processar a entrada de dados do usuário. Uma das atividades mais comuns que os usuários efetuarão dentro de seus programas é dar um clique duplo no mouse ou em uma opção. Um clique duplo é uma série de dois cliques de um botão no mouse, o segundo ocorrendo após um tempo especificado após o primeiro. Algumas vezes, você poderá querer controlar a duração do tempo que deixará passar antes de um segundo clique, porém ainda contar os dois cliques como um clique duplo. Seus programas podem usar a função *SetDoubleClickTime* para controlar a velocidade do clique duplo do mouse. A função *SetDoubleClickTime* define o tempo do clique duplo para o mouse. O tempo do clique duplo é o número máximo de milissegundos que pode ocorrer entre o primeiro e o segundo clique de um clique duplo. Seus programas usarão a função *SetDoubleClickTime* como mostrado no seguinte protótipo:

```
BOOL SetDoubleClickTime(
   UINT  uIntervalo          // intervalo do clique duplo
);
```

O parâmetro *uIntervalo* especifica o número de milissegundos que pode ocorrer entre o primeiro e o segundo clique de um clique duplo. Se *uIntervalo* for deixado como zero, o Windows usará o tempo normal de 500 milissegundos para o clique duplo.

Nota: A função SetDoubleClickTime altera o tempo do clique duplo para todas as janelas no sistema. Se seus programas alterarem o valor do tempo do clique duplo, deverão retornar para seu estado original quando o programa terminar.

Exatamente como seus programas podem definir a quantidade de tempo entre dois cliques do mouse que qualificam como um clique duplo, assim também pode o seu programa recuperar o tempo do duplo clique para o mouse. A função *GetDoubleClickTime* recupera o tempo atual do clique duplo. Como você sabe, o clique duplo é uma série de dois cliques do botão do mouse, o segundo clique ocorrendo dentro de um tempo especificado após o primeiro. Da mesma forma, o tempo do clique duplo é o número máximo de milissegundos que pode ocorrer entre o primeiro e o segundo clique de um clique duplo. Seus programas implementarão a função *GetDoubleClickTime* de acordo com a seguinte forma generalizada:

```
UINT GetDoubleClickTime(void)
```

Se a função *GetDoubleClickTime* for bem-sucedida, o valor de retorno especificará o tempo atual do clique duplo em milissegundos. Para compreender melhor o processamento que as funções *GetDoubleClickTime* e *SetDoubleClickTime* executam, considere o programa *Get_Set.cpp*, contido no CD-ROM que acompanha este livro.

Toda vez que o usuário der um clique duplo no mouse, o programa exibirá uma caixa de mensagem dizendo que recebeu o clique duplo. No entanto, cada vez que o usuário selecionar o menu *Testar!*, o programa aumentará o tempo do clique duplo em um décimo de segundo. Observe que, ao terminar, o programa restaura o tempo do clique duplo ao seu valor original.

1344 PERMUTANDO OS BOTÕES DO MOUSE

Como você aprendeu na dica anterior, algumas vezes seus programas precisam modificar a duração entre os cliques em um clique duplo. Similarmente, algumas vezes, dentro de seus programas, o usuário precisará inverter os botões do mouse — usando o botão direito para efetuar as ações do botão esquerdo e usando o botão esquerdo para efetuar as ações do botão direito. A função *SwapMouseButton* inverte ou restaura o significado dos botões esquerdo e direito do mouse. O Windows fornece a inversão dos botões como uma conveniência para as pessoas que usam o mouse com a mão esquerda. Normalmente, apenas o Painel de Controle chama a função *SwapMouseButton*, embora um aplicativo possa livremente chamar a função. Se seus programas precisarem usar *SwapMouseButton*, você a chamará como mostrado no protótipo a seguir:

```
BOOL SwapMouseButton(
   BOOL fSwap     // inverte ou restaura os botões
);
```

O parâmetro *fSwap* especifica se o significados dos botões do mouse estão atualmente invertidos ou restaurados. Se *fSwap* for *True*, o botão esquerdo gerará mensagens do botão direito, e o botão direito gerará mensagens do botão esquerdo. Se *fSwap* for *False*, os botões serão restaurados aos seus significados originais. O CD-ROM que acompanha este livro inclui o programa *Swap_B.cpp*, que permuta o estado dos botões do mouse toda vez que o usuário seleciona a opção *Testar!* no menu.

Nota: O mouse é um recurso compartilhado, e inverter o significado de seus botões afeta todos os aplicativos. Sempre que possível, seus programas devem evitar inverter os botões do mouse.

1345 DETERMINANDO SE O USUÁRIO PRESSIONOU UMA TECLA

Seus programas podem efetuar processamento toda vez que o usuário pressiona uma tecla ou dá um clique no botão do mouse. No entanto, alguns aplicativos podem requerer que o usuário efetue vários passos para selecionar uma opção. Por exemplo, um programa poderia responder diferentemente se o usuário pressionasse a tecla de função F12, e, depois, desse um clique com o mouse em uma opção que o programa responderá se o usuário

simplesmente der um clique do mouse na opção. Para efetuar esse processamento assíncrono (isto é, não ao mesmo tempo), seus programas podem usar a função *GetAsyncKeyState*. A função *GetAsyncKeyState* determina se uma tecla está liberada ou pressionada no tempo em que o programa chama a função e se o usuário pressionou a tecla após uma chamada anterior a *GetAsyncKeyState*. Você implementará a função *GetAsyncKeyState* como mostrado aqui:

```
SHORT GetAsyncKeyState(
    int vTecla          // código de tecla virtual
);
```

O parâmetro *vTecla* especifica um dos 256 códigos possíveis de tecla virtual, como já detalhou a Dica 1340. Se a função for bem-sucedida, o valor de retorno especificará se o usuário pressionou a tecla desde a última chamada à *GetAsyncKeyState* e se a tecla está atualmente liberada ou pressionada. Se o bit mais significativo está ligado, a tecla está pressionada, e, se o bit menos significativo está ligado, o usuário pressionou a tecla após a chamada anterior a *GetAsyncKeyState*. O valor de retorno é zero se uma janela em outro encadeamento ou processo atualmente tem o foco do teclado.

Nota: Sob o Windows 95 você pode usar as constantes de código de tecla virtual VK_SHIFT, VK_CONTROL e VK_MENU, como valores para o parâmetro vTecla. Isso dá o estado das teclas Shift, Ctrl ou Alt sem distinguir entre esquerda e direita.

Nota: Sob o Windows NT, você pode usar as seguintes constantes de tecla virtual como valores para vTecla para distinguir entre as ocorrências esquerda e direita dessas teclas:

VK_LSHIFT	VK_RSHIFT
VK_LCONTROL	VK_RCONTROL
VK_LMENU	VK_RMENU

Essas constantes que distinguem entre esquerda e direita somente estão disponíveis quando você chama as funções *GetKeyboardState*, *SetKeyboardState*, *GetAsyncKeyState*, *GetKeyState* e *MapVirtualKey*.

A função *GetAsyncKeyState* trabalha com os botões do mouse. No entanto, ela checa no estado dos botões físicos do mouse, não nos botões lógicos para os quais os botões físicos estão mapeados. Por exemplo, a chamada *GetAsyncKeyState(VK_LBUTTON)* sempre retorna o estado do botão físico esquerdo do mouse, que o sistema pode ter mapeado para o botão lógico esquerdo ou direito do mouse. Você pode determinar o mapeamento atual do sistema dos botões físicos com os botões lógicos do mouse chamado *GetSystemMetrics(SM_SWAPBUTTON)*, que retorna *True* se o sistema operacional inverteu anteriormente os botões do mouse, e *False*, em caso contrário. Para compreender melhor o processamento que *GetAsyncKeyState* executa, considere o programa *Get_Async_Key.cpp* contido no CD-ROM que acompanha este livro. O programa *Get_Async_Key.cpp* retorna o status atual das teclas Shift quando o usuário seleciona o item de menu *Testar!*

APRESENTANDO AS BARRAS DE ROLAGEM 1346

Uma janela em um aplicativo pode exibir muitas coisas. Ela pode exibir um objeto de dados, tal como um documento ou uma imagem gráfica, que é maior que a área cliente da janela. Quando a janela tiver uma barra de rolagem, o usuário poderá rolar o objeto de dados dentro da área cliente para visualizar o documento ou a imagem gráfica por inteiro. Um aplicativo deve incluir uma barra de rolagem em uma janela sempre que o conteúdo da área cliente for maior que o tamanho da área cliente da janela. Uma barra de rolagem normal tem três componentes: as setas em ambos os sentidos, a largura ou altura da barra de rolagem e o *elevador*, o quadradinho que o usuário arrasta dentro da barra. Por exemplo, a Figura 1346 mostra um programa com uma barra de rolagem nas bordas vertical e horizontal, bem como os componentes da barra de rolagem.

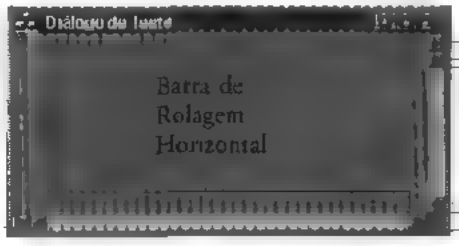

Figura 1346 Uma janela com duas barras de rolagem internas e seus componentes.

1347 COMPREENDENDO OS DIFERENTES TIPOS DE BARRA DE ROLAGEM

Como você aprendeu na dica anterior, toda barra de rolagem, independentemente de seu posicionamento ou direção, compartilha certos aspectos comuns com outras barras de rolagem padrão. No entanto, como você viu na Figura 1346, existem pelo menos dois tipos diferentes de barra de rolagem. Existem na verdade duas categorias de barras de rolagem. A primeira categoria é a das *barras de rolagem da área cliente*. As barras de rolagem da área cliente podem se estender verticalmente ou horizontalmente e podem aparecer como uma barra de rolagem normal ou como uma barra de trilhos. A segunda categoria é a das barras de rolagem que não são de área cliente. O Windows anexa as barras de rolagem que não são da área cliente à borda da janela imediatamente após você tornar as barras visíveis.

Como você sabe, é possível criar barras de rolagem que não são da área cliente para seu aplicativo dentro da estrutura *WNDCLASS* que seus programas passam para a função *CreateWindow*. Quando você anexa uma barra de rolagem à borda de uma janela, o Windows automaticamente subtrai a largura da barra de rolagem da área cliente da janela para que a pintura dentro da área cliente não invada as barras de rolagem. Seus programas também podem chamar a função *ShowScrollBar* para anexar barras de rolagem à borda interna de uma janela. Você aprenderá mais sobre *ShowScrollBar* na dica a seguir.

Além de anexar barras de rolagem à borda interna das janelas, o sistema operacional automaticamente anexará barras de rolagem às caixas de lista e caixas de combinação quando a lista de itens exceder o tamanho da janela de lista. Seus programas também podem anexar barras de rolagem às caixas de ediçao. No entanto, as caixas de edição de uma única linha suportam apenas a barra de rolagem horizontal, enquanto as caixas de edição de múltiplas linhas podem suportar tanto barra de rolagem horizontal quanto vertical.

1348 USANDO A FUNÇÃO SHOWSCROLLBAR

Como visto na dica anterior, seus programas podem anexar barras de rolagem às janelas tanto na criação da janela quanto após a criação da janela. Para anexar ou ocultar uma barra de rolagem em uma janela criada anteriormente, seus programas usarão a função *ShowScrollBar*. A função *ShowScrollBar* mostra ou oculta a barra de rolagem que você especificar. Você usará a função *ShowScrollBar* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL ShowScrollBar(
   HWND  hJan,     // indicativo da janela com a barra de rolagem
   int wBarra,     // sinalizador de barra de rolagem
   BOOL bMostrar   // sinalizador de visibilidade da barra de rolagem
);
```

O parâmetro *hJan* identifica um controle de barra de rolagem ou uma janela com uma barra de rolagem padrão, dependendo do valor do parâmetro *wBarra*. O parâmetro *bMostrar* especifica se o Windows mostra ou oculta a barra de rolagem. Se *bMostrar* for *True*, o Windows mostrará a barra de rolagem. Caso contrário, ele ocultará a barra de rolagem. O parâmetro *wBarra* especifica as barras de rolagem que o Windows mostrará ou ocultará. O parâmetro *wBarra* pode ser um dos valores listados na Tabela 1348.1.

Tabela 1348.1 Valores possíveis para o parâmetro ***wBarra***.

Valor	Significado
SB_BOTH	Mostra ou oculta as barras de rolagem horizontal e vertical padrão de uma janela.
SB_CTL	Mostra ou oculta um controle de barra de rolagem. O parâmetro *hJan* precisa ser o indicativo do controle de barra de rolagem.
SB_HORZ	Mostra ou oculta as barras de rolagem horizontais padrão de uma janela.
SB_VERT	Mostra ou oculta a barra de rolagem vertical de uma janela.

*Nota: Você não deve chamar a função **ShowScrollBar** para ocultar uma barra de rolagem enquanto estiver processando uma mensagem da barra de rolagem.*

Após exibir as barras de rolagem, você frequentemente desejará controlar como o usuário poderá manipular as barras de rolagem. Um dos controles mais comuns que você aplicará a uma barra de rolagem é habilitar ou desabilitar uma ou ambas as setas dentro da barra de rolagem. Para fazer isso, você chamará a função *EnableScrollBar*. A função *EnableScrollBar* habilita ou desabilita uma ou ambas as setas da barra de rolagem. Você implementará a função *EnableScrollBar* dentro de seus programas, como mostrado aqui:

```
BOOL EnableScrollBar(
   HWND hJan,    // indicativo da janela ou da barra de rolagem
   UINT wszBR,   // sinalizador do tipo da barra de rolagem
   UINT wSetas   // sinalizador de seta da barra de rolagem
);
```

O parâmetro *hJan* identifica uma janela ou um controle de barra de rolagem, dependendo do valor do parâmetro *wszBR*. O parâmetro *wszBR* especifica o tipo da barra de rolagem. Esse parâmetro pode ser um dos valores detalhados na Tabela 1348.1.

O parâmetro *wSetas* especifica se as setas da barra de rolagem estão habilitadas ou desabilitadas e indica quais setas estão habilitadas ou desabilitadas. O parâmetro *wSetas* pode ser um dos valores listados na Tabela 1348.2.

Tabela 1348.2 Valores possíveis para o parâmetro ***wSetas***.

Valor	Significado
ESB_DISABLE_BOTH	Desabilita ambas as setas em uma barra de rolagem.
ESB_DISABLE_DOWN	Desabilita a seta para baixo em uma barra de rolagem vertical.
ESB DISABLE_LEFT	Desabilita a seta para a esquerda em uma barra de rolagem horizontal.
ESB_DISABLE_LTUP	Desabilita a seta para a esquerda em uma barra de rolagem horizontal ou a seta para cima de uma barra de rolagem vertical.
ESB DISABLE RIGHT	Desabilita a seta para a direita em uma barra de rolagem horizontal.
ESB_DISABLE_RTDN	Desabilita a seta para a direita em uma barra de rolagem horizontal ou a seta para baixo de uma barra de rolagem vertical.
ESB_DISABLE_UP	Desabilita a seta para cima em uma barra de rolagem vertical.
ESB_ENABLE_BOTH	Habilita ambas as setas em uma barra de rolagem.

Somente com essas funções seus programas podem executar a maioria das manipulações de barra de rolagem de que precisarem. O CD-ROM que acompanha este livro inclui o programa *Scroll1.cpp*, que usa ambas as funções *ShowScrollBars* e *EnableScrollBars* para controlar uma barra de rolagem dentro de seu programa.

COMPREENDENDO A POSIÇÃO E A FAIXA DA BARRA DE ROLAGEM 1349

Como você viu na dica anterior, acrescentar uma barra de rolagem em suas janelas é um processo simples. Quando você cria uma barra de rolagem, o intervalo padrão de valores que os dois extremos do controle representam é de 0 a 100. Na maioria dos casos, um aplicativo modificará o intervalo para refletir o tamanho do documento ou da imagem. A posição do elevador é o valor dentro do intervalo onde o elevador está localizado. Por exemplo, se o intervalo for de 0 a 100 e a posição do elevador for 50, o elevador aparecerá no centro do controle.

Você também pode definir o *tamanho da página* para os controles da sua barra de rolagem. O tamanho da página representa o número de incrementos dentro do intervalo de rolagem que a página poderá exibir em um momento. Por exemplo, se o intervalo for de 0 a 100 e o tamanho da página for definido como 50, a página poderá exibir metade do intervalo do controle em um determinado momento. Seus programas podem usar as funções *SetScrollInfo* e *GetScrollInfo* para definir e recuperar o intervalo de uma barra de rolagem, a posição do elevador e o tamanho da página.

1350 COMPREENDENDO AS MENSAGENS DA BARRA DE ROLAGEM

Em dicas anteriores você aprendeu alguns dos fundamentos sobre as barras de rolagem. À medida que o usuário usa a barra de rolagem para rolar, a barra gera mensagens para os seus aplicativos, exatamente como o mouse e o teclado. Quando o usuário dá um clique no mouse em uma barra de rolagem, o Windows envia uma mensagem *WM_HSCROLL* ou uma mensagem *WM_VSCROLL* ao aplicativo, dependendo se a barra de rolagem é horizontal ou vertical. A palavra menos significativa do parâmetro *lParam* informa a função de processamento do lugar onde o mouse estava quando o usuário deu um clique no botão do mouse. Dependendo de onde o mouse estava localizado ao tempo do clique, o Windows enviará uma dentre dez mensagens aos seus programas. A Figura 1350 mostra as posições possíveis para o clique do mouse e a mensagem correspondente.

Figura 1350 Algumas das mensagens que o Windows gera para cliques do mouse na barra de rolagem.

Quando o usuário soltar o botão do mouse após dar um clique em algum lugar em uma barra de rolagem, o Windows enviará a mensagem *SB_ENDSCROLL* para o aplicativo, a não ser que o usuário estivesse movendo o elevador com o mouse. Nesse caso, o Windows gerará a mensagem *SB_THUMBPOSITION* quando o usuário soltar o botão do mouse.

1351 OBTENDO AS DEFINIÇÕES ATUAIS DA BARRA DE ROLAGEM

As barras de rolagem iniciam em certas posições padrão quando você as inclui em janelas dentro de seus programas. No entanto, como também aprendeu, seus programa frequentemente alterarão as definições de uma barra de rolagem durante o processamento de um programa. Seus programas, com frequência, precisam efetuar processamento específico com base nas definições da barra de rolagem. Para ajudar seus programas, a determinar as definições atuais de uma barra de rolagem, seus programas podem usar a função *GetScrollInfo*. A função *GetScrollInfo* obtém os parâmetros de uma barra de rolagem, incluindo as posições mínima e máxima de rolagem, o tamanho da página e a posição do elevador. Você usará a função *GetScrollInfo* dentro de seus programas como mostrado no protótipo a seguir:

```
BOOL GetScrollInfo(
    HWND hjan,          // indicativo da janela com a barra de rolagem
    int fnBarra,        // sinalizador de barra de rolagem
    LPSCROLLINFO lpsi   // ponteiro para estrutura para os
                        // parâmetros da rolagem
);
```

Como você pode ver, a função *GetScrollInfo* aceita três parâmetros. O parâmetro *lpsi*, sobre o qual você aprenderá mais tarde nesta dica, retorna um valor do tipo *SCROLLINFO*, uma estrutura definida pelo sistema com os membros mostrados aqui

```
typedef struct tagSCROLLINFO
 {
   UINT cbSize;
   UINT fMask;
   int  nMin;
   int  nMax;
   UINT nPage;
   int nPos;
```

```
    int nTrackPos;
} SCROLLINFO;
```

Tanto *GetScrollInfo* quanto *SetScrollInfo* usam a estrutura *SCROLLINFO*. A Tabela 1351.1 detalha os membros da estrutura e seus usos.

Tabela 1351.1 Os membros da estrutura ***SCROLLINFO***.

Membro	Descrição
cbSize	Especifica o tamanho, em bytes, da estrutura *SCROLLINFO*
fMask	Especifica os parâmetros da barra de rolagem para definir ou ler. Esse membro pode ser uma combinação dos valores detalhados pela Tabela 1351.2
nMin	Especifica a posição mínima de rolagem.
nMax	Especifica a posição máxima de rolagem.
nPage	Especifica o tamanho da página. Uma barra de rolagem usa esse valor para determinar o tamanho apropriado para o elevador.
nPos	Especifica a posição do elevador.
nTrackPos	Especifica a posição imediata de um elevador que o usuário está arrastando. Um aplicativo pode ler esse valor ao processar a mensagem de notificação *SB_THUMBTRACK*. Um aplicativo não pode definir a posição de rolagem imediata. A função *SetScrollInfo* ignora esse membro.

Como você aprendeu na Tabela 1351.1, o membro *fMask* pode ter um dentre vários valores predefinidos. A Tabela 1351.2 mostra os valores possíveis para o membro *fMask*.

Tabela 1351.2 Os valores predefinidos possíveis para o membro ***fMask***.

Valor	Significado
SIF_ALL	Combinação de *SIF_PAGE*, *SIF_POS* e *SIF_RANGE*.
SIF_DISABLENOSCROLL	Seus programas usarão esse valor somente ao definir os parâmetros de uma barra de rolagem. Se os novos parâmetros da barra de rolagem tornarem a barra desnecessária, você precisará desabilitar a barra, em vez de removê-la.
SIF_PAGE	O membro *nPage* contém o tamanho da página para uma barra de rolagem proporcional.
SIF_POS	O membro *nPos* contém a posição da caixa de rolagem (ou elevador).
SIF_RANGE	Os membros *nMin* e *nMax* contêm os valores mínimo e máximo para o intervalo de rolagem.

Dentro da função *GetScrollInfo*, o parâmetro *hwnd* identifica um controle de barra de rolagem ou uma janela com uma barra de rolagem padrão, dependendo do valor do parâmetro *fnBar*. O parâmetro *fnBar* especifica o tipo de barra de rolagem para o qual serão lidos os parâmetros. O parâmetro *fnBar* pode ser um dos valores detalhados na Tabela 1351.3.

Tabela 1351.3 Valores possíveis para o parâmetro ***fnBar***.

Valor	Significado
SB_CTL	Lê os parâmetros para um controle de barra de rolagem. O parâmetro *hwnd* precisa ser o indicativo do controle de barra de rolagem.
SB_HORZ	Lê os parâmetros para a barra de rolagem horizontal padrão da janela dada.
SB_VERT	Lê os parâmetros para a barra de rolagem vertical padrão da janela dada.

Finalmente, o parâmetro *lpsi* aponta para uma estrutura *SCROLLINFO* cujo membro *fMask*, na entrada da função, especifica os parâmetros da barra de rolagem a serem lidos. Antes de retornar, a função copia os parâmetros especificados nos membros apropriados da estrutura. O membro *fMask* pode ser uma combinação dos valores detalhados na Tabela 1351.4 (observe que a função *GetScrollInfo* aceita somente certos valores para o membro *fMask*).

Tabela 1351.4 Valores possíveis para o membro ***fMask*** da estrutura ***SCROLLINFO***.

Valor	Significado
SIF_PAGE	Copia a página de rolagem para o membro *nPage* da estrutura *SCROLLINFO* para a qual o parâmetro *lpsi* aponta.
SIF_POS	Copia a posição de rolagem para o membro *nPos* da estrutura *SCROLLINFO* para a qual o parâmetro *lpsi* aponta.
SIF_RANGE	Copia o intervalo de rolagem para os membros *nMin* e *nMax* da estrutura *SCROLLINFO* para a qual o parâmetro *lpsi* aponta.

A função *GetScrollInfo* permite que os aplicativos usem posições de rolagem de 32 bits. Embora as mensagens que indicam a posição da barra de rolagem, *WM_HSCROLL* e *WM_VSCROLL*, forneçam somente 16 bits de dados de posição, as funções *SetScrollInfo* e *GetScrollInfo* fornecem 32 bits de dados de posição da barra de rolagem. Portanto, um aplicativo pode chamar *GetScrollInfo* enquanto processa a mensagem *WM_HSCROLL* ou *WM_VSCROLL* para obter dados de posição de 32 bits da barra de rolagem.

A limitação em acessar dados de posição de barra de rolagem de 32 bits aplica-se à rolagem em tempo real do conteúdo de uma janela. Um aplicativo implementa a rolagem em tempo real processando as mensagens *WM_HSCROLL* ou *WM_VSCROLL* que carregam o valor de notificação *SB_THUMBTRACK*, desse modo controlando a posição do elevador à medida que o usuário o mover. Infelizmente, não há uma função que obtenha a posição do elevador em 32 bits à medida que o usuário o move. Como *GetScrollInfo* fornece somente a posição estática, um aplicativo somente pode obter dados de posição de 32 bits antes ou após uma operação de rolagem.

1352 ROLANDO O CONTEÚDO DA JANELA

A função mais importante que suas barras de rolagem executarão é permitir que os usuários rolem o conteúdo de uma janela. Você controlará a rolagem da sua janela com a função *ScrollWindowEx*. A função *ScrollWindowEx* rola o conteúdo da área cliente da janela especificada. Você usará a função *ScrollWindowEx* dentro de seus programas, como mostra o protótipo a seguir:

```
int ScrollWindowEx(
   HWND hJan,      // indicativo da janela a rolar
   int dx,         // quantidade da rolagem horizontal
   int dy,         // quantidade de rolagem vertical
   CONST RECT *prcRola,  // endereço da estrutura com retângulo de rolagem
   CONST RECT *prcCorte, // endereço da estrutura com retângulo de corte
   HRGN hrgnAtual,    // indicativo da região de atualização
   LPRECT prcAtual,  // endereço da estrutura para o retângulo de atualização
   UINT sinaliz       // sinalizadores de rolagem
);
```

Como você pode ver, *ScrollWindowEx* recebe oito parâmetros, a maioria dos quais é previsível. A Tabela 1352.1 detalha os parâmetros para a função *ScrollWindowEx*.

Tabela 1352.1 Os parâmetros para a função ***ScrollWindowEx***.

Parâmetro	Descrição
hJan	Identifica a janela onde a função ***ScrollWindowEx*** deverá rolar a área cliente.
dx	Especifica a quantidade, em unidades do dispositivo, da rolagem horizontal. Esse parâmetro precisa ser um valor negativo para rolar para a esquerda.
dy	Especifica a quantidade, em unidades do dispositivo, da rolagem vertical. Esse parâmetro precisa ser um valor negativo para rolar para cima.
prcRola	Aponta para a estrutura *RECT* que especifica a porção da área cliente que a função *ScrollWindowEx* deverá rolar. Se esse parâmetro for *NULL*, a função *ScrollWindowEx* rolará toda a área cliente.
prcCorte	Aponta para a estrutura *RECT* que contém as coordenadas do retângulo de corte A função *ScrollWindowEx* somente afeta os bits do dispositivo dentro do retângulo de corte. O Windows pintará as partes que a função rolar de fora do retângulo para o interior; o Windows não pintará as partes que a função rolar do interior do retângulo para fora.
hrgnAtual	Identifica a região que a função *ScrollWindowEx* deve modificar para conter a região que a rolagem invalida. O parâmetro *hrgnAtual* pode ser *NULL*.
prcAtual	Aponta para uma estrutura *RECT* que recebe os limites do retângulo que a rolagem invalida. O parâmetro *prcAtual* pode ser *NULL*.
sinaliz	Especifica sinalizadores que controlam a rolagem. Esse parâmetro tem um dos valores detalhados na Tabela 1352.2.

Como indica a Tabela 1352.1, o parâmetro *sinaliz* pode ter um dos vários valores predefinidos. A Tabela 1352.2 lista os possíveis valores para o parâmetro ***sinaliz***.

Tabela 1352.2 Valores possíveis para o parâmetro ***sinaliz***.

Valor	Significado
SW_ERASE	Apaga a região recém-invalidada enviando uma mensagem *WM_ERASEBKGND* para a janela quando você incluir o sinalizador *SW_ERASE* com o sinalizador *SW_INVALIDATE*.
SW_INVALIDATE	Invalida a região que o parâmetro *hrgnAtual* identifica após a rolagem.
SW_SCROLLCHILDREN	Rola todas as janelas-filha dentro do retângulo para o qual o parâmetro *prcRola* aponta. O Windows rola as janelas-filha o número de pixels especificados pelos parâmetros *dx* e *dy*. O Windows envia uma mensagem *WM_MOVE* para todas as janelas-filha afetadas pelo retângulo *prcRola*, mesmo que elas não se movam.

Se a função for bem-sucedida, o valor de retorno será *SIMPLEREGION* (região retangular invalidada), *COMPLEXREGION* (região invalidada não-retangular; retângulos sobrepostos), ou *NULLREGION* (nenhuma região invalidada) Se a função falhar, ela retornará o valor *ERROR*.

Se a chamada de função não especificar os sinalizadores *SW_INVALIDATE* e *SW_ERASE*, *ScrollWindowEx* não invalida a área a partir da qual rola. Se um desses sinalizadores estiver ligado, *ScrollWindowEx* invalidará a área a partir da qual irá rolar. O Windows não atualizará a área até que o aplicativo chame a função *UpdateWindow*, chame a função *RedrawWindow* (especificando o sinalizador *RDW_UPDATENOW* ou *RDW_ERASENOW*), ou receba a mensagem *WM_PAINT* da fila de mensagem.

Se a janela tiver o estilo *WS_CLIPCHILDREN*, as áreas retornadas que *hrgnAtual* e *prcAtual* especificam, representarão a área total da janela rolada que o Windows precisa atualizar, incluindo quaisquer áreas em janelas-filha. Se a chamada à *ScrollWindowEx* especificar o sinalizador *SW_SCROLLCHILDREN*, o Windows não atualizará corretamente a tela se o usuário rolar uma seção de uma janela-filha. O Windows não apaga a seção da janela-filha rolada que está fora do retângulo-fonte e não redesenha corretamente a janela-filha em seu novo destino. Para mover as janelas-filha que não estão completamente dentro do retângulo que *prcRola* especifica, use

a função *DeferWindowPos*. O Windows reposicionará o cursor se o sinalizador *SW_SCROLLCHILDREN* estiver ligado e o retângulo do ponto de inserção passar pelo retângulo de rolagem.

O Windows determina todas as coordenadas de entrada e de saída (para *prcRola*, *prcCorte*, *prcAtual* e *hrgnAtual*) como coordenadas cliente, independentemente se a janela tem ou não o estilo de classe *CS_OWNDC* ou *CS_CLASSDC*. Use as funções *LPtoDP* e *DPtoLP* para converter de/para coordenadas lógicas, se necessário.

Para compreender melhor o processamento que a função *ScrollWindowEx* executa, considere o programa *Scroll Window.cpp*, contido no CD-ROM que acompanha este livro. Esse programa permite que o usuário acrescente uma linha de cada vez à janela do programa. Após o número de linhas exceder a quantidade de espaço disponível dentro da janela, a barra de rolagem se tornará ativa. Quando o usuário der um clique com o mouse na seta para cima ou para baixo, o elevador se moverá na direção apropriada.

1353 COMPREENDENDO A MENSAGEM WM_SIZE

Na dica anterior, você aprendeu sobre o programa *Scroll_Window*, que permite que o usuário selecione a barra de rolagem para mover uma janela para cima e para baixo. Como você aprendeu, o programa somente ativou a barra de rolagem quando o conteúdo era grande demais para exibir dentro da janela. Se você experimentar com o programa *Scroll_Window* e tornar a janela maior, verá que a barra de rolagem fica inativa novamente após você deixar a janela grande o suficiente para exibir o texto. O programa *Scroll_Window.cpp* captura a mensagem do sistema *WM_SIZE* para ajustar a barra de rolagem após você redefinir a janela, como mostrado aqui:

```
// Toda vez que a janela tem seu tamanho ajustado,
// recalcula o número de linhas
// A área cliente pode exibir e definir a barra de
// rolagem apropriadamente
Case WM_SIZE :
    {
       RECT rect;
       GetClientRect(hWnd, &rect);

       nDspLines = rect.bottom / 20;
       if (nDspLines < nNumItems)
        {
          SCROLLINFO si;

          si.cbSize = sizeof(SCROLLINFO);
          si.fMask  = SIF_POS | SIF_RANGE | SIF_PAGE;
          si.nMin   = 0;
          si.nMax   = nNumItems-1;
          si.nPage  = nDspLines;
          si.nPos   = nCurPos;
          EnableScrollBar(hWnd, SB_VERT, ESB_ENABLE_BOTH);
                SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
          }
          else
             EnableScrollBar(hWnd, SB_VERT, ESB_DISABLE_BOTH);
        }
       break;
```

No caso do programa *Scroll_Windows.cpp*, o comando *case WM_SIZE* testa o número de linhas que a janela pode exibir com relação ao número de linhas que está exibindo atualmente. Se o número de linhas possíveis for maior que o número de linhas mostradas, o programa desabilitará as barras de rolagem. Caso contrário, ele mudará o tamanho do elevador (se necessário) e exibirá a barra de rolagem de tamanho recém-ajustado.

Seus programas usarão a mensagem *WM_SIZE* para verificar o tamanho atual da janela e substituir quaisquer itens dentro da janela cuja posição dependa do tamanho da janela. Por exemplo, se você tiver uma caixa de edição na janela que exiba um título e o usuário redefinir a janela, você poderá querer redefinir o tamanho da caixa de texto concorrentemente com a janela. A ação que seus programas executarão quando receberem uma mensagem *WM_SIZE* diferirá, com base no tipo de redefinição de tamanho que o usuário faça. A mensagem *WM_SIZE* passa uma constante que representa o tipo de ajuste dentro do parâmetro *wParam*. A Tabela 1353 detalha os valores possíveis para o parâmetro *wParam*.

Tabela 1353 Valores possíveis para *wParam* com uma mensagem *WM_SIZE*

Valor	Significado
SIZE_MAXHIDE	O Windows envia a mensagem para todas as janelas de ativação instantânea quando o usuário maximizar alguma outra janela
SIZE_MAXIMIZED	O usuário maximizou a janela.
SIZE_MAXSHOW	O Windows envia a mensagem para todas as janelas de ativação instantânea quando a unidade restaurar alguma outra janela ao seu tamanho anterior.
SIZE_MINIMIZED	O usuário minimizou a janela.
SIZE_RESTORED	O usuário reajustou o tamanho da janela, mas nem o valor *SIZE_MINIMIZED* nem *SIZE_MAXIMIZED* se aplica.

Adicionalmente, a mensagem *WM_SIZE* passa informações sobre a nova janela dentro do parâmetro *lParam*. A palavra menos significativa de *lParam* especifica a nova largura da área cliente, e a palavra mais significativa especifica a nova altura da área cliente.

COMPREENDENDO A MENSAGEM WM_PAINT 1354

Como você aprendeu na dica anterior, seus aplicativos receberão a mensagem *WM_SIZE* cada vez que o usuário redefinir a janela do aplicativo. O Windows também enviará aos seus aplicativos a mensagem *WM_PAINT* seguindo a mensagem *WM_SIZE* cada vez que o usuário redefinir o tamanho da janela do aplicativo. Na verdade, seu aplicativo receberá a mensagem *WM_PAINT* quando o Windows ou outro aplicativo fizer uma solicitação para pintar uma porção da janela do seu aplicativo. O Windows envia a mensagem *WM_PAINT* quando seu programa chama a função *UpdateWindow* ou *RedrawWindow*, ou envia a função *DispatchMessage* quando seu aplicativo usa a função *GetMessage* ou *PeekMessage* para obter uma mensagem *WM_PAINT*.

O parâmetro *wParam* para a mensagem *WM_PAINT* contém o valor *hdc*, um indicativo para um dispositivo do contexto. Ele identifica o dispositivo do contexto no qual o Windows desenha. Se o parâmetro *wParam* for *NULL*, o aplicativo deverá usar o dispositivo do contexto padrão (em vez de criar um dispositivo de contexto privado). Alguns controles comuns usam *wParam* para habilitar o desenho em um dispositivo do contexto diferente do dispositivo do contexto padrão. Outras janelas podem ignorar com segurança *wParam*. Você aprenderá mais sobre os dispositivos do contexto em dicas posteriores.

A função *DefWindowProc* valida a região de atualização. A função também poderá enviar a mensagem *WM_NCPAINT* para o procedimento de janela se o Windows precisar pintar a moldura da janela e adicionalmente enviará a mensagem *WM_ERASEBKGND* se o Windows precisará apagar o fundo da janela.

O sistema enviará a mensagem *WM_PAINT* quando não houver outras mensagens na fila de mensagem do aplicativo. *DispatchMessage* determina para onde enviar a mensagem; *GetMessage* determina qual mensagem despachar. *GetMessage* retorna a mensagem *WM_PAINT* quando não existem outras mensagens na fila de mensagem do aplicativo, e *DispatchMessage* envia a mensagem para o procedimento de janela apropriado.

Uma janela pode receber mensagens internas de pintura como resultado de chamar *RedrawWindow* com o sinalizador *RDW_INTERNALPAINT* ligado. Nesse caso, a janela pode não ter uma região de atualização. Um aplicativo deverá chamar a função *GetUpdateRect* para determinar se a janela tem uma região de atualização. Se *GetUpdateRect* retornar zero, o aplicativo não deverá chamar as funções *BeginPaint* e *EndPaint*. Se o aplicativo não chamar a função *GetUpdateRect*, seções da janela poderão não ser atualizadas corretamente.

Um aplicativo precisa examinar suas estruturas de dados internas para cada mensagem *WM_PAINT* para checar qualquer pintura interna necessária, porque uma região de atualização não-*NULL* ou uma chamada a *RedrawWindow* com o sinalizador *RDW_INTERNALPAINT* ligado pode ter causado uma mensagem *WM_PAINT*.

O Windows envia uma mensagem *WM_PAINT* interna somente uma vez. Após *GetMessage* retornar uma mensagem *WM_PAINT* ou *UpdateWindow* enviar *PeekMessage* para uma janela, o Windows não encaminhará ou enviará outras mensagens *WM_PAINT* até que o programa invalide a janela ou até que o programa chame *RedrawWindow* novamente com o sinalizador *RDW_INTERNALPAINT* ligado.

Para alguns controles comuns, o processamento padrão da mensagem *WM_PAINT* checa o parâmetro *wParam*. Se *wParam* não for *NULL*, os controles assumirão que o valor será um indicativo para um dispositivo do contexto e usarão esse dispositivo do contexto para pintar. Para compreender melhor o processamento que

seus programas efetuam quando recebem uma mensagem *WM_PAINT*; veja o programa *Scroll_Window.cpp*, já apresentado na dica anterior.

1355 OUTRAS MENSAGENS DA BARRA DE ROLAGEM QUE SEUS PROGRAMAS PODEM CAPTURAR

A barra de rolagem gera diferentes mensagens específicas, dependendo de onde o usuário der o clique dentro da área da barra de rolagem. Além das mensagens *WM_SIZE* e *WM_PAINT* sobre as quais você aprendeu nas Dicas 1353 e 1354, seus programas também devem capturar as mensagens *WM_VSCROLL* e *WM_HSCROLL*.

O Windows envia a mensagem *WM_VSCROLL* para uma janela quando um evento de rolagem ocorre na barra de rolagem vertical padrão da janela. O Windows também envia a mensagem *WM_VSCROLL* para o proprietário de um controle de barra de rolagem vertical quando um evento de rolagem ocorre no controle. Quando seu programa recebe uma mensagem *WM_VSCROLL*, a palavra menos significativa de *wParam* especifica um valor de barra de rolagem que indica a solicitação de rolagem do usuário. A palavra menos significativa pode ser um dos valores listados na Tabela 1355.1.

Tabela 1355.1 Valores possíveis para a palavra menos significativa do parâmetro ***wParam***.

Valor	Significado
SB_BOTTOM	Rola para a inferior direita.
SB_ENDSCROLL	Finaliza a rolagem.
SB_LINEDOWN	Rola uma linha para baixo.
SB_LINEUP	Rola uma linha para cima.
SB_PAGEDOWN	Rola uma página para baixo.
SB_PAGEUP	Rola uma página para cima.
SB_THUMBPOSITION	Rola para a posição absoluta (um deslocamento numérico exato a partir do início da janela, tal como "12 linhas para baixo a partir do topo"). O parâmetro *nPos* especificará a posição atual.
SB_THUMBTRACK	Arrasta a caixa de rolagem para a posição específica. O parâmetro *nPos* especifica a posição atual.
SB_TOP	Rola para a superior esquerda.

Além do valor que o Windows passa dentro da palavra menos significativa do parâmetro *wParam*, seus programas também devem verificar a palavra mais significativa de *wParam*. A palavra mais significativa especificará a posição atual do elevador se o parâmetro *nScrollCode* for *SB_THUMBPOSITION* ou *SB_THUMBTRACK*; caso contrário, a palavra mais significativa irá conter um valor não-usável. Finalmente, o parâmetro *lParam* contém o indicativo de controle da barra de rolagem. Se uma barra de rolagem não envia a mensagem, *lParam* é *NULL*.

Os aplicativos que fornecem retroalimentação à medida que o usuário arrasta o elevador tipicamente usam a mensagem de notificação *SB_THUMBTRACK*. Se um aplicativo rola o conteúdo da janela, ele também precisa usar a função *SetScrollPos* para reicinializar a posição do elevador.

O Windows envia a mensagem *WM_HSCROLL* para uma janela quando um evento de rolagem ocorre na barra de rolagem horizontal padrão. O Windows também envia a mensagem *WM_HSCROLL* para o proprietário de um controle de barra de rolagem horizontal quando um evento de rolagem ocorre no controle. Exatamente como a mensagem *WM_VSCROLL* inclui informações adicionais dentro dos parâmetros *wParam* e *lParam*, assim também acontece com a mensagem *WM_HSCROLL*.

Com *WM_HSCROLL*, a palavra menos significativa de *wParam* especifica um valor de barra de rolagem que indica a solicitaçao de rolagem do usuário. A palavra menos significativa de *wParam* pode conter um dos valores listados na Tabela 1355.2.

Tabela 1355.2 Valores possíveis para a palavra menos significativa do parâmetro *wParam*.

Valor	Significado
SB_BOTTOM	Rola para a direita inferior.
SB_ENDSCROLL	Finaliza a rolagem.
SB_LINELEFT	Rola para a esquerda uma unidade.
SB_LINERIGHT	Rola para a direita uma unidade.
SB_PAGELEFT	Rola para a esquerda pela largura da janela.
SB_PAGERIGHT	Rola para a direita pela largura da janela.
SB_THUMBPOSITION	Rola para a posição absoluta (um deslocamento numérico exato a partir do lado esquerdo da janela). O parâmetro *nPos* (a palavra mais significativa de *wParam*) especifica a posição atual.
SB_THUMBTRACK	Arrasta a caixa de rolagem para a posição específica. O parâmetro *nPos* (a palavra mais significativa de *wParam*) especifica a posição atual.
SB_TOP	Rola para a esquerda superior.

Exatamente como com a mensagem *WM_VSCROLL*, a palavra mais significativa de *wParam* especifica a posição atual do elevador se o parâmetro *nScrollCode* é *SB_THUMBPOSITION* ou *SB_THUMBTRACK*; caso contrário, a palavra mais significativa não contém um valor usável. O parâmetro *lParam* retornará o indicativo do controle da barra de rolagem se uma barra de rolagem enviar a mensagem. Se uma barra de rolagem não enviar a mensagem, *hwndScrollBar* será *NULL*.

Observe que ambas as mensagens *WM_HSCROLL* e *WM_VSCROLL* carregam somente 16 bits de dados de posição da caixa de rolagem. Portanto, os aplicativos que dependem unicamente de *WM_HSCROLL* e *WM_VSCROLL* para dados de posição de rolagem têm uma valor de posição prático máximo de 65.535. No entanto, como as funções *SetScrollPos*, *SetScrollRange*, *GetScrollPos* e *GetScrollRange* suportam dados de posição de barra de rolagem de 32 bits, há um modo de contornar a barreira de 16 bits das mensagens *WM_HSCROLL* e *WM_VSCROLL* dentro dessas funções. Veja no arquivo de ajuda do compilador maiores informações sobre como contornar a barreira de 16 bits.

HABILITANDO E DESABILITANDO AS BARRAS DE ROLAGEM 1356

Seus programas podem ter controle completo sobre as barras de rolagem. Uma das atividades mais comuns que seus programas executarão com as barras de rolagem será habilitá-las e desabilitá-las. Como você viu na Dica 1352, por exemplo, seus programas poderão desabilitar a barra de rolagem se o usuário ajustar o tamanho da janela para um tamanho grande o suficiente para exibir tudo que a janela contém. Para habilitar e desabilitar as barras de rolagem, seus programas usarão a função *EnableScrollBar*. A função *EnableScrollBar* habilita ou desabilita uma ou ambas as setas da barra de rolagem. Você usará a função *EnableScrollBar* dentro de seus programas, como mostra o protótipo a seguir:

```
BOOL EnableScrollBar(
   HWND hJan,      // indicativo do programa ou barra de rolagem
   UINT wBRflags, // sinalizador do tipo da barra de rolagem
   UINT wSetas     // sinalizador da seta da barra de rolagem
);
```

Como você pode ver, a função *EnableScrollBar* aceita três parâmetros. O parâmetro *hJan* identifica uma janela ou um controle de barra de rolagem, dependendo do valor do parâmetro *wBRflags*. O parâmetro *wBRflags* especifica o tipo da barra, e pode ter um dos valores mostrados na Tabela 1356.1

Tabela 1356.1 Valores possíveis para o parâmetro *wBRflags*

Valor	Significado
SB_BOTH	Habilita ou desabilita as setas nas barras de rolagem horizontal e vertical associadas com a janela especificada O parâmetro *hJan* precisa ser o indicativo da janela
SB CTL	Identifica a barra de rolagem como um controle de barra de rolagem. O parâmetro *hJan* precisa ser o indicativo para o controle de barra de rolagem.
SB_HORZ	Habilita ou desabilita as setas na barra de rolagem horizontal associada com a janela especificada. O parâmetro *hJan* precisa ser o indicativo da janela.
SB_VERT	Habilita ou desabilita as setas na barra de rolagem vertical associada com a janela especificada. O parâmetro *hJan* precisa ser o indicativo da janela.

Finalmente, o parâmetro *wSetas* especifica se as setas de barra de rolagem estão habilitadas ou desabilitadas e indica qual seta a função *EnableScrollBar* deve habilitar ou desabilitar. O parâmetro *wSetas* pode ser um dos valores mostrados na Tabela 1356.2.

Tabela 1356.2 Valores possíveis para o parâmetro *wSetas*.

Valor	Significado
ESB_DISABLE_BOTH	Desabilita ambas as setas em uma barra de rolagem.
ESB_DISABLE_DOWN	Desabilita a seta para baixo em uma barra de rolagem vertical.
ESB_DISABLE_LEFT	Desabilita a seta para a esquerda em uma barra de rolagem horizontal.
ESB_DISABLE_LTUP	Desabilita a seta para a esquerda em uma barra de rolagem horizontal ou a seta para cima de uma barra vertical.
ESB_DISABLE_RIGHT	Desabilita a seta para a direita em uma barra de rolagem horizontal.
ESB_DISABLE_RTDN	Desabilita a seta para a direita em uma barra de rolagem horizontal ou a seta para baixo de uma barra de rolagem vertical.
ESB_DISABLE_UP	Desabilita a seta para cima em uma barra de rolagem vertical.
ESB_ENABLE_BOTH	Habilita ambas as setas em uma barra de rolagem

Se a função habilitar ou desabilitar com sucesso as setas que o parâmetro *wSetas* especificar, o valor de retorno será diferente de zero. Se as setas já estiverem no estado solicitado ou um erro ocorrer, o valor de retorno será zero.

O CD-ROM que acompanha este livro inclui o programa *Enable_Disable.cpp*, que habilita e desabilita a barra de rolagem vertical dependendo do tamanho e do conteúdo da janela. Você deve observar dentro da função *ProcJan* como o programa *Enable_Disable.cpp* verifica o tamanho e o conteúdo da janela toda vez que a janela recebe um comando *WM_SIZE* e habilita ou desabilita as barras de rolagem, conforme apropriado, após a janela receber e o programa processar o comando *WM_SIZE*.

1357 USANDO A FUNÇÃO SCROLLDC

Nas dicas anteriores, você aprendeu sobre diversas funções da barra de rolagem e as mensagens que elas geram. Embora você ainda não tenha aprendido sobre os dispositivos do contexto (aprenderá em dicas posteriores), é importante compreender como seus programas podem rolar as janelas que exibem gráficos ou outros itens não-textuais. Como regra, quando você desenhar itens não-textuais em uma janela, usará um *dispositivo do contexto* para fazer isso. Você também poderá usar um dispositivo do contexto quando desenhar texto em uma janela, embora geralmente usará a indireção para acessar o contexto. Quando você rolar um dispositivo do contexto dentro de uma janela, usará o processamento ligeiramente diferente para rolar o dispositivo do contexto ou uma parte do dispositivo do contexto. A função *ScrollDC* rola um retângulo de bits dentro de um dispositivo do contexto horizontalmente e verticalmente. Você usará a função *ScrollDC* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL ScrollDC(
   HDC hDC,        // indicativo do dispositivo do contexto
   int dx,         // unidades de rolagem horizontal
   int dy,         // unidades de rolagem vertical
   CONST RECT *lprcRola,   // endereço da estrutura para o
                           // retângulo de rolagem
   CONST RECT *lprcCorta, // endereço da estrutura para o
                           // retângulo de corte
   HRGN hrgnAtual,         // indicativo da região de rolagem
   LPRECT lprcAtual        // endereço da estrutura para o
);                         // retângulo de atualização
```

A função *ScrollDC* aceita os parâmetros detalhados na Tabela 1357.

Tabela 1357 Os Parâmetros para a função ***ScrollDC***.

Parâmetro	Descrição
hDC	Identifica o dispositivo do contexto que contém os bits que a função *ScrollDC* deverá rolar.
dx	Especifica a quantidade, em unidades de dispositivo, da rolagem horizontal Esse parâmetro precisa ser um valor negativo para rolar para a esquerda.
dy	Especifica a quantidade, em unidades de dispositivo, da rolagem vertical. Esse parâmetro precisa ser um valor negativo para rolar para cima.
lprcRola	Aponta para a estrutura *RECT* que contém as coordenadas do retângulo de rolagem.
lprCorte	Aponta para a estrutura *RECT* que contém as coordenadas do retângulo de corte. A função *ScrollDC* afeta somente os bits do dispositivo dentro do retângulo de corte. O Windows pintará bits que a função rolar de fora do retângulo para o interior; ele não pintará os bits que a função rolar do interior do retângulo para fora.
hrgnAtual	Identifica a região que o processo de rolagem descobre. *ScrollDC* define essa região; não é necessariamente um retângulo
lprcAtual	Aponta para a estrutura *RECT* que recebe as coordenadas do retângulo que delimita a região de atualização de rolagem. Essa é a maior área retangular que requer repintura. Quando a função retorna, os valores na estrutura estão em coordenadas clientes, independentemente do modo de mapeamento para o dispositivo do contexto especificado Isso permite que os aplicativos usem a região de atualização em uma chamada à função *InvalidateRgn*, se necessário.

Se o parâmetro *lprcUpdate* for *NULL*, o Windows não calculará o retângulo de atualização. Se ambos os parâmetros *hrgnAtual* e *lprcAtual* forem *NULL*, o Windows não calculará a região de atualização. Se *hrgnUpdate* não for *NULL*, o Windows procederá como se ela contivesse um indicativo válido da região que o processo de rolagem (que *ScrollDC* define) descobre. Para compreender melhor o processamento que ScrollDC efetua, considere o programa *Scroll_DC* no CD-ROM que acompanha este livro. O programa *Scroll_DC* desenha uma série de linhas na janela. Quando você, então, tentar rolar a janela com a barra de rolagem, o programa *Scroll_DC* rolará somente uma seção da janela.

Embora o código do programa *Enable_Disable.cpp* provavelmente não seja claro, pois você ainda não aprendeu sobre os dispositivos do contexto, o processamento final que a função executa (na verdade rolando a porção da janela) é simples. O programa *Scroll_DC* gera um dispositivo do contexto, depois define a região que *ScrollDC* moverá para 20 unidades menores que a área cliente da janela. O programa então chama a função *ScrollDC* para rolar o retângulo para a direita.

1358 COMPREENDENDO O MODELO DE MEMÓRIA WIN32

Como você pode ter imaginado a partir de dicas em seções anteriores, o modelo de memória do Win32 simplifica grandemente o gerenciamento da memória. Os modelos de memória Small, Large e Huge não existem mais no Windows de 32 bits. Como não há modelos de memória, os programas Win32 não distinguem entre memória *near* e *far*. Sem segmentos, o programa e seus dados agora residem na mesma memória linear, o que facilita o tratamento dos blocos de programa e de dados muito grandes.

No ambiente Win32, cada processo tem seu próprio espaço de endereço virtual de 32 bits de até quatro gigabytes (4GB). O Windows torna disponível os dois gigabytes na memória baixa (*0x00000000* até *0x7FFFFFFF*) para o usuário e reserva os dois gigabytes na memória alta (*0x80000000* até *0xFFFFFFFF*) para o núcleo do computador. Os endereços que os processos usam não representam mais posições físicas na memória. Em vez disso, o núcleo do sistema operacional (o software que controla a CPU, a memória, os encadeamentos etc.) mantém um mapa da página para cada processo, que o Windows usa para traduzir endereços virtuais em endereços físicos correspondentes. Um processo não pode escrever fora de seu próprio espaço de processo (o que protege um processo do outro).

A API Win32 suporta as funções de alocação *GlobalAlloc* e *LocalAlloc*. No ambiente Win32, as alocações globais e locais são efetivamente a mesma coisa. No ambiente Win16, a alocação local estava dentro do espaço de endereço para o processo, e a alocação global estava fora do espaço do endereço para o processo. No ambiente Win32, o Windows aloca ambos os tipos de memória dentro do espaço de endereço para o processo, e ambos os tipos de memória são acessíveis a partir de dentro de seus programas que usam ponteiros de 32 bits. No entanto, como você verá, usar alocação local pode tornar seus programas mais fáceis de compreender

O ambiente Win32 introduz dois novos modos de seus programas gerenciarem a memória, o *gerenciador de memória virtual* e o *gerenciador do heap local*. As funções da API do gerenciador de memória virtual são similares às funções da API do gerenciador da memória global, exceto que seus programas podem reservar grandes blocos de memória virtual, e, depois, alocar esses blocos de memória virtual reservada mais tarde. O novo gerenciador do heap difere dos gerenciadores de heap que você usou em dicas anteriores em que o novo gerenciador de heap permite que seus programas criem múltiplos heaps separados. Os heaps múltiplos lhe oferecem um modo simples e eficiente de alocar quantidades pequenas de memória (tal como a memória que um ponteiro simples requer). Nas próximas dicas, você aprenderá mais sobre o gerenciamento de memória do Windows. Você também aprenderá sobre algumas das funções que seus programas podem usar para gerenciar mais eficientemente a memória virtual e a global do Windows.

1359 COMPREENDENDO A MEMÓRIA GLOBAL E A LOCAL

Já vimos na dica anterior que, no modelo de memória linear de 32 bits dos aplicativos Windows, o Windows (e, portanto, seus programas) não distingue entre memória global e local. Como resultado, o Windows (e, portanto, seus programas) não distingue entre heap global e local. Portanto, os objetos de memória que seus programas alocam com chamadas à *GlobalAlloc* e *LocalAlloc* são os mesmos. O Windows aloca a memória em páginas privadas e comprometidas (isto é, páginas de memória não-acessíveis por outros programas) com acesso leitura/gravação. A *memória privada* é memória que outros programas, mesmo programas de execução concorrente, não podem acessar.

As funções *GlobalAlloc* e *LocalAlloc* podem alocar um bloco de memória de qualquer tamanho que 32 bits possam representar e podem caber dentro da memória disponível (incluindo a armazenagem disponível dentro do *arquivo de paginação*, sobre o qual você aprenderá mais em dicas posteriores). Os objetos de memória que seus programas alocam podem ser fixos (em posições específicas de memória) ou móveis. Se você alocar um objeto de memória fixa, ele permanecerá dentro de sua dada posição na memória física durante toda a sua vida.

Você também pode marcar os objetos móveis como descartáveis. No Windows 3.x, os objetos de memória móvel eram importantes para o gerenciamente da memória. Por outro lado, no Win32, seus programas usarão a memória virtual. O sistema pode, portanto, gerenciar memória sem causar impacto no endereço virtual da memória. Quando o sistema mover uma página de memória, o Windows simplesmente mapeará a página virtual para o processo para a nova posição. Seus programas usarão memória móvel para alocar memória descartável, que seus programas usarão com freqüência e descartarão regularmente (tal como matrizes, ponteiros etc.).

COMPREENDENDO A MEMÓRIA VIRTUAL 1360

As dicas anteriores apresentaram o conceito de memória virtual. Como indicou a dica anterior, o gerenciador de memória virtual permite que seus programas tratem a quantidade limitada de memória física do computador e o tamanho (geralmente maior) do arquivo de paginação como um único bloco de memória contígua. Para permitir que seus programas gerenciem a memória desse modo, o gerenciador de memória virtual permite que seus programas trabalhem com *mapas* para a memória real, em vez de trabalhar com a memória real. Como você está trabalhando com a ilusão da memória contígua, em vez de um bloco de memória realmente contíguo, os projetistas do Windows chamaram o modelo de memória de *modelo de memória virtual*. A Figura 1360 mostra um modelo lógico de como o Windows usa a memória virtual para acessar a memória física.

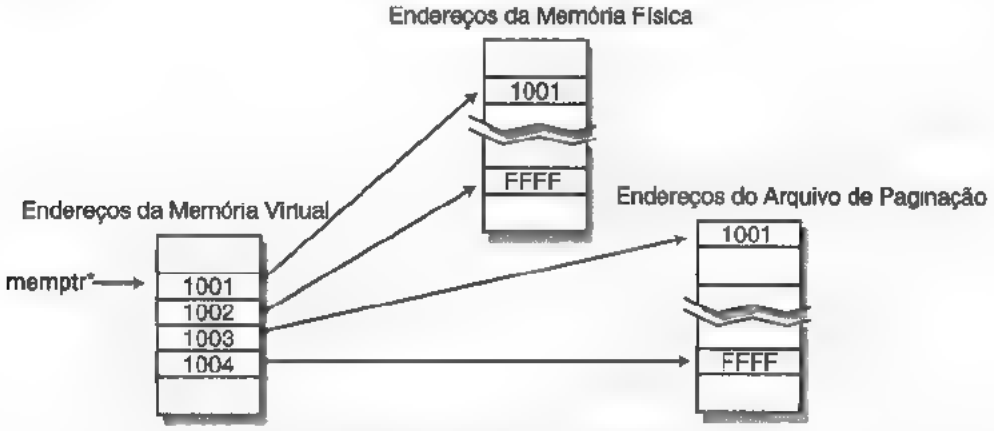

Figura 1360 O Windows usa a memória virtual para acessar a memória física.

Devido aos modos como o Windows gerencia a memória virtual, o espaço do endereço virtual para cada processo é muito maior que o endereço físico total disponível para todos os processos. Para aumentar o tamanho da memória, o Windows usa o disco rígido como armazenagem adicional. A quantidade total de memória disponível para todos os processos é a soma da memória física e o espaço livre disponível no disco dentro do *arquivo de paginação* do Windows. O arquivo de paginação é um arquivo no disco que o Windows usa para aumentar a memória efetiva do seu computador. O Windows organiza o espaço de memória virtual em *páginas*, ou unidades de memória. O tamanho da página depende do computador hospedeiro. Você pode chamar a função *GetSystemInfo* para determinar o tamanho da página para um computador. Nos computadores x86, o tamanho da página é 4Kb.

Como você aprendeu, no modelo de memória Win32, o sistema operacional fornece a cada processo seu próprio espaço privado de endereço. Quando um encadeamento em um processo estiver em execução, esse encadeamento poderá acessar somente a memória que pertence ao seu processo. A memória que pertence a todos os outros processos está oculta e inacessível para o encadeamento em execução. Como cada processo tem seu próprio 4Gb virtual de espaço de endereço, todo processo vê a memória como se ela fosse do endereço 0x00000000 até 0xFFFFFFFF. Dois processos que rodem simultaneamente podem cada um, armazenar memória no endereço 0x12341234 sem interferir um com o outro. Claramente, esse compartilhamento de endereço não seria possível se ambos os processos estivessem usando o mesmo espaço de endereço.

Na realidade, o gerenciador da memória virtual mapeia o endereço virtual para um endereço físico real — o que pode estar na memória física do computador ou pode estar no arquivo de paginação. A consideração importante é que, no que interessa ao aplicativo, o endereço da memória é 0x12341234 — esteja a memória na posição da RAM 0x00012345 ou no setor 14352 no disco rígido do computador. Portanto, o gerenciador de memória virtual lhe permite executar múltiplos programas simultaneamente sem garantir que cada programa não anule a memória dos outros programas em execução.

No Windows 95, o sistema operacional divide o endereço virtual de 4Gb para cada processo em quatro partições. O Windows 95 usa a partição de *0x00000000* até *0x003FFFFF* (uma partição de 4Mb na parte inferior do espaço de endereço virtual) para manter compatibilidade com o MS-DOS e com o Windows de 16 bits. Seus aplicativos Win32 não devem ler ou gravar nessa partição. Se seus aplicativos tentarem acessar essa me-

mória, o sistema operacional retornará um ponteiro *NULL* e poderá causar instabilidade (resultando no travamento do programa, bloqueio do sistema operacional e assim por diante).

O Windows 95 usa os 2Gb restantes do espaço do endereço virtual de cada processo para armazenar arquivos compartilhados e arquivos do sistema operacional. Quando você trabalhar com memória dentro do espaço de endereço do seu programa, deverá evitar acesso do espaço do endereço acima de *0x80000000* a não ser que seu programa esteja tentando especificamente acessar um arquivo compartilhado ou do sistema.

1361 REVISITANDO OS HEAPS

Como visto na dica anterior, o Windows aloca em seus programas 4Gb de espaço de endereço virtual para cada processo em execução. As funções de heap Win32 permitem que um processo crie um *heap privado*, que é um bloco de uma ou mais páginas no espaço de endereço do processo. A função *HeapCreate* cria um heap de um determinado tamanho, e as funções *HeapAlloc* e *HeapFree* alocam e liberam memória no heap. Quando você criar heaps dentro de seus programas Windows, o Windows criará o heap iniciando em *0x7FFFFF*, e o heap crescerá para baixo na memória reservada do processo de 2Gb.

*Os objetos do heap poderão crescer dinamicamente dentro do intervalo que você especificar na criação de*les com a função *HeapCreate*. O tamanho máximo do heap determina o número de páginas de memória que o heap reserva. O tamanho inicial determina o número de páginas de leitura/gravação comprometido que o Windows inicialmente alocará para o heap. O Windows automaticamente aloca páginas adicionais a partir do espaço reservado se os requisitos de *HeapAlloc* excederem o tamanho atual das páginas comprometidas. Após o Windows comprometer as páginas para um heap, ele não liberará essas páginas comprometidas até que o processo termine ou que o programa destrua o heap com a função *HeapDestroy*. Como a memória que seus programas alocam no heap com *HeapAlloc* tem uma posição fixa dentro do espaço de endereço virtual do processador e o sistema não pode compactar o heap, você sempre deve escrever seus aplicativos de tal forma que eles minimizem a fragmentação do heap

A memória de um heap privado é acessível somente para o processo que criou o heap. Se uma biblioteca de ligações dinâmicas (DLL) criar um heap privado, o Windows criará esse heap privado no espaço de endereço do processo que chamou a biblioteca de ligações dinâmicas (sob o Windows 95, acima de 0x80000000). No entanto, somente o processo que chamou a biblioteca de ligações dinâmicas pode acessar as informações no heap privado criado pela biblioteca de ligações dinâmicas — o que significa que múltiplos processos que executam a mesma biblioteca de ligações dinâmicas podem resultar em múltiplos heaps privados criados por biblioteca de ligações dinâmicas, mas cada ocorrência de biblioteca somente pode acessar um único heap privado.

1362 ALOCANDO UM BLOCO DE MEMÓRIA A PARTIR DO HEAP GLOBAL

Seus programas podem usar diversas técnicas para alocar diferentes tipos de memória dentro dos ambientes Windows 95 e Windows NT. Uma das alocações mais comum que você efetuará será alocar memória a partir do heap global — que é similar à alocação na qual você usou funções, tais como *malloc* e *halloc*, para efetuar em programas do DOS. Você usará a função *GlobalAlloc* para alocar memória a partir do heap global. A função *GlobalAlloc* alocará o número de bytes que você especificar no heap. Você usará a função *GlobalAlloc* como mostrado no protótipo a seguir:

```
HGLOBAL GlobalAlloc(
   UINT uFlags,          // atributos de alocação do objeto
   DWORD dwBytes,        // número de bytes a alocar
);
```

O parâmetro *uFlags* especifica como alocar a memória. Se *uFlags* especificar zero, o padrão será o sinalizador *GMEM_FIXED*. Exceto pelas combinações incompatíveis que as tabelas a seguir observam especificamente, seus programas podem usar qualquer combinação dos sinalizadores que as tabelas a seguir detalham quando seus programas chamam *GlobalAlloc*. Para indicar se a função aloca memória fixa ou móvel, especifique um dos sinalizadores da Tabela 1362.1.

Tabela 1362.1 Tipos de alocação de memória para uso com ***GlobalAlloc***.

Sinalizador	Significado
GMEM_FIXED	Aloca memória fixa. Seus programas não podem combinar esse sinalizador com o sinalizador *GMEM_MOVEABLE* ou *GMEM_DISCARDABLE*. O valor de retorno é um ponteiro para o bloco de memória. Para acessar a memória, o processo de chamada simplesmente converte o valor de retorno para um ponteiro.
GMEM_MOVEABLE	Aloca memória móvel. Seus programas não podem combinar esse sinalizador com *GMEM_FIXED*. O valor de retorno é o indicativo do objeto de memória. O indicativo é uma quantidade de 32 bits e é privado para o processo chamador. Para traduzir o indicativo em um ponteiro, use a função *GlobalLock*.
GPTR	Combina o sinalizador *GMEM_FIXED* e *GMEM_ZEROINIT* da Tabela 1362.2.
GHND	Combina o sinalizador *GMEM_MOVEABLE* e o sinalizador *GMEM-ZEROINIT* da Tabela 1362.2

Além dos sinalizadores especificados na Tabela 1362.1, o parâmetro *uFlags* pode conter qualquer um dos valores detalhados na Tabela 1362.2, exceto onde a tabela observa que há um sinalizador especificamente em conflito com outro sinalizador.

Tabela 1362.2 Valores de sinalizador adicionais para a função ***GlobalAlloc***.

Sinalizador	Significado
GMEM_DDESHARE	Aloca memória que as funções do intercâmbio dinâmico de dados (DDE) usarão para um diálogo do intercâmbio dinâmico de dados. Esse sinalizador está disponível por questões de compatibilidade com os aplicativos Win16. Alguns aplicativos poderão usar *GMEM_DDESHARE* para aumentar o desempenho das operações de intercâmbio dinâmico de dados, e seus programas deverão especificar o sinalizador *GMEM_DDESHARE* se forem usar a memória para intercâmbio dinâmico de dados. Somente os processos que usam o intercâmbio dinâmico de dados ou a área de transferência para comunicações interprocesso devem especificar o sinalizador *GMEM_DDESHARE*.
GMEM_DISCARDABLE	Aloca memória descartável (memória que não está fixa em um endereço específico dentro do espaço de endereço virtual do processo). Seus programas não podem combinar esse sinalizador com o sinalizador *GMEM_FIXED*. Alguns aplicativos com base no Win32 podem ignorar esse sinalizador.
GMEM_LOWER	O Win32 ignora esse sinalizador. A API Win32 fornece esse sinalizador somente para compatibilidade com o Windows versão 3.*x*.
GMEM_NOCOMPACT	Não compacta ou descarta a memória para satisfazer a solicitação de alocação
GMEM_NODISCARD	Não descarta a memória para satisfazer a solicitação de alocação.
GMEM_NOT_BANKED	O Win32 ignora esse sinalizador. A API Win32 fornece esse sinalizador somente para compatibilidade com o Windows versão 3.x.
GMEM_NOTIFY	O Win32 ignora esse sinalizador. A API Win32 fornece esse sinalizador somente para compatibilidade com o Windows versão 3.x.
GMEM_SHARE	Aloca memória que as funções do intercâmbio dinâmico de dados (DDE) usarão para um diálogo DDE. É o mesmo que o sinalizador *GMEM_DDESHARE*.
GMEM_ZEROINIT	Inicializa o conteúdo da memória com zero.

Além de especificar o tipo de memória para *GlobalAlloc* alocar, seus programas precisam especificar o parâmetro *dwBytes*, que especifica o número de bytes a alocar. Se esse parâmetro for zero e o parâmetro *uFlags* especificar o sinalizador *GMEM_MOVEABLE*, a função retornará um indicativo para um objeto de memória que o Windows marca como descartado. Se a função for bem-sucedida, o valor de retorno será o indicativo do objeto de memória recém-alocado. Se a função falhar, o valor de retorno será *NULL*.

Se o heap não contiver espaço livre suficiente para satisfazer a solicitação, *GlobalAlloc* retornará *NULL*. Como *GlobalAlloc* usa *NULL* para indicar um erro, o Windows nunca aloca o endereço virtual zero. É, portanto, fácil detectar o uso de um ponteiro *NULL*. O Windows cria toda a memória com acesso de execução. Ele não requer uma função especial para executar código gerado dinamicamente. O Windows garante que a memória que seus programas alocam com a função *GlobalAlloc* estará alinhada em um limite de 8 bytes.

O Windows limita as funções *GlobalAlloc* e *LocalAlloc* para um total combinado de 65 536 indicativos para *GMEM_MOVEABLE* (para memória alocada globalmente) e *LMEM_MOVEABLE* (para memória alocada localmente) por processo. Essa limitação não se aplica à memória *GMEM_FIXED* ou *LMEM_FIXED*. Se a função GlobalAlloc for bem-sucedida, ela alocará pelo menos a quantidade de memória que a chamada da função requer; o processo poderá, no entanto, usar a quantidade inteira. Para determinar o número real de bytes que a função *GlobalAlloc* alocou, use a função *GlobalSize*.

Para compreender melhor o processamento que a função *GlobalAlloc* executa, considere o programa *Global_Alloc*, contido no CD-ROM que acompanha este livro. O programa *Global_Alloc* aloca memória para armazenar uma string. Quando o aplicativo inicia, a rotina de tratamento da mensagem *WM_CREATE* cria um buffer de 27 caracteres (26 bytes e um finalizador *NULL*). Quando o usuário seleciona *Testar!*, o programa exibe o buffer e seu tamanho na tela.

1363 USANDO GLOBALREALLOC PARA ALTERAR DINAMICAMENTE OS TAMANHOS DO HEAP

Seus programas podem usar a função *GlobalAlloc* para alocar memória a partir do heap global. Frequentemente, no entanto, seus programas precisam realocar um bloco de memória após sua alocação inicial. Você pode fazer isso com a função *GlobalReAlloc*. A função *GlobalReAlloc* altera o tamanho ou atributos de um objeto de memória global especificado. O tamanho pode aumentar ou diminuir, dependendo da chamada. Você usará a função *GlobalReAlloc* dentro de seus programas, como mostra o protótipo a seguir:

```
HGLOBAL GlobalReAlloc(
   HGLOBAL hMem,    // indicativo para o objeto de memória global
   DWORD dwBytes,   // novo tamanho do bloco
   UINT uFlags      // como realocar o objeto
);
```

O indicativo *hMem* identifica o objeto de memória global que *GlobalReAlloc* deverá realocar. A função *GlobalAlloc* ou *GlobalReAlloc* terá anteriormente retornado esse indicativo. O parâmetro *dwBytes* especifica o novo tamanho, em bytes, do bloco de memória. Se o parâmetro *dwBytes* for zero e o parâmetro *uFlags* especificar o sinalizador *GMEM_MOVEABLE*, a função retornará o indicativo de um objeto de memória que o Windows marca como descartado. Se o parâmetro *dwBytes* for zero e *uFlags* especificar o sinalizador *GMEM_MODIFY*, a função da API ignorará o parâmetro *dwBytes*. O parâmetro *uFlags* especifica como realocar o objeto de memória global. Se o parâmetro *uFlags* especificar o sinalizador *GMEM_MODIFY*, o parâmetro *uFlags* modificará os atributos de um objeto de memória, e a função da API ignorará o parâmetro *dwBytes*. Caso contrário, o parâmetro *uFlags* controlará a realocação do objeto de memória.

Quando você chamar a função *GlobalReAlloc*, seu programa poderá combinar o sinalizador *GMEM_MODIFY* com um ou ambos os sinalizadores detalhados na Tabela 1363.1.

Tabela 1363.1 Sinalizadores compatíveis com ***GMEM_MODIFY***.

Sinalizador	Significado
GMEM_DISCARDABLE	Aloca memória descartável se você também especifica o sinalizador *GMEM_MODIFY*. O Windows ignora esse sinalizador a não ser que você tenha anteriormente alocado o objeto como móvel ou a não ser que também especifique o sinalizador *GMEM_MOVEABLE*.
GMEM_MOVEABLE	*Windows* NT *somente*. Muda um objeto de memória fixa para um objeto de memória móvel se você também especifica o sinalizador *GMEM_MODIFY*.

Se o parâmetro *uFlags* não especificar *GMEM_MODIFY*, ele poderá ser qualquer combinação dos sinalizadores listados na Tabela 1363.2.

Tabela 1363.2 Sinalizadores adicionais para o parâmetro ***uFlags***.

Sinalizador	Significado
GMEM_MOVEABLE	Se *dwBytes* for zero, *GMEM_MOVEABLE* descartará um bloco de memória anteriormente móvel e descartável Se o contador de bloqueios do objeto não for zero ou se o bloco não for móvel ou descartável, a função falhará. Se *dwBytes* for diferente de zero, *GMEM_MOVEABLE* habilitará o sistema a mover o bloco realocado para uma nova posição sem alterar o atributo de móvel ou fixo do objeto de memória. Se o objeto for fixo, o indicativo que a função retorna poderá ser diferente do indicativo que o parâmetro *hMem* especifica. Se o objeto for móvel, seus programas poderão mover o bloco sem invalidar o indicativo do objeto, mesmo se uma chamada anterior à função *GlobalLock* estiver atualmente bloqueando o objeto. Para obter o novo endereço do bloco de memória, use *GlobalLock*
GMEM_NOCOMPACT	Impede que o Windows compacte ou descarte a memória para satisfazer a solicitação de alocação.
GMEM_ZEROINIT	Faz o Windows inicializar o conteúdo da memória adicional como zero se o objeto de memória está crescendo em tamanho.

Se a função for bem-sucedida, o valor de retorno será o indicativo do objeto de memória realocado. Se a função falhar, o valor de retorno será *NULL*. Se *GlobalReAlloc* realocar um objeto móvel, o valor de retorno será o indicativo do objeto de memória. Para converter o indicativo em um ponteiro, use a função *GlobalLock*. Se *GlobalReAlloc* realocar um objeto fixo, o valor do indicativo retornado será o endereço do primeiro byte do bloco de memória. Para acessar a memória, um processo pode simplesmente converter o valor de retorno em um ponteiro. Se *GlobalAlloc* falha, ela não libera a memória original, e o indicativo e o ponteiro originais ainda são válidos.

Para compreender melhor o processamento que *GlobalReAlloc* executa, considere o programa *Global_ReAlloc* contido no CD-ROM que acompanha este livro. O programa *Global_ReAlloc* efetua processamento similar ao programa *Global_Alloc*. No entanto, o programa *Global_ReAlloc* também realoca outros 27 bytes de memória para mostrar o alfabeto minúsculo quando o usuário seleciona a opção *Testar!*

DESCARTANDO UM BLOCO DE MEMÓRIA ALOCADO 1364

Em dicas anteriores, você usou as funções-membro *free* e *delete* para descartar memória alocada dentro de seus programas. Quando você alocar ou realocar memória a partir de um heap global, seus programas deverão usar a função *GlobalDiscard* para liberar essa memória após seu programa completar o processamento na memória. A função *GlobalDiscard* descarta um bloco de memória global que você alocou anteriormente com o sinalizador *GMEM_DISCARDABLE* ligado. O contador de bloqueio do objeto de memória que você quer descartar, precisará ser zero, ou a função não funcionará para descartar a memória. Você usará a função *GlobalDiscard*, como segue:

```
HGLOBAL GlobalDiscard(
   HGLOBAL hglbMem   // indicativo para objeto de memória global
);
```

O parâmetro *hglbMem* identifica o objeto de memória global a descartar. Se a função for bem-sucedida, a função retornará o indicativo do objeto de memória (isto é, o indicativo *hglbMem*). Se a função falhar, a função retornará o valor *NULL*.

A função *GlobalDiscard* descarta somente os objetos globais que o processo chamador alocou com o sinalizador *GMEM_DISCARDABLE*. Se um processo tentar descartar um objeto fixo ou bloqueado, a função não funcionará. Embora a função *GlobalDiscard* descarte o bloco de memória do objeto, o indicativo do objeto permanece válido. Um processo pode subsequentemente passar o indicativo para a função *GlobalReAlloc* para alocar outro bloco de memória global que o mesmo indicativo identifica.

1365 USANDO A FUNÇÃO GLOBALFREE

Na dica anterior, você aprendeu sobre a função *GlobalDiscard*, que seus programas usarão para descartar um bloco de memória anteriormente alocado e para manter o indicativo para esse bloco de memória para possível uso futuro. Por outro lado, se você sabe que seu programa não reutilizará o mesmo bloco de memória, se você quiser manter seu programa usando o mesmo bloco de memória, ou se você não tiver certeza se seu programa alocou o bloco com o sinalizador *GMEM_DISCARDABLE*, seus programas poderão usar a função *GlobalFree* para liberar um objeto de memória. A função *GlobalFree* libera o objeto de memória global que você especifica e invalida o indicativo do objeto de memória. Você usará a função *GlobalFree* como mostrado no protótipo a seguir:

```
HGLOBAL  GlobalFree(HGLOBAL hMem);
```

O parâmetro *hMem* identifica o objeto de memória global a liberar. Ao contrário da função *GlobalDiscard*, se a função *GlobalFree* for bem-sucedida, o valor de retorno será *NULL*. Se a função *GlobalFree* falhar, o valor de retorno será igual ao indicativo do objeto de memória global.

A corrupção do heap ou uma exceção de violação de acesso (*EXCEPTION_ACCESS_VIOLATION*) poderá ocorrer se o processo tentar examinar ou modificar a memória após o processo ter liberado anteriormente a memória. Se o parâmetro *hglbMem* for *NULL*, *GlobalFree* falhará, e o sistema gerará uma exceção de violação de acesso. Tanto *GlobalFree* quanto *LocalFree* liberarão um objeto de memória bloqueado. O objeto de memória bloqueado tem um contador de bloqueio maior que zero. A função *GlobalLock* bloqueia um objeto de memória global (o que evita que outra função descarte o objeto de memória) e incrementa em um o contador de bloqueio do objeto. Para ler o contador de bloqueio de um objeto de memória global, use a função *GlobalFlags*.

*Nota: Se um aplicativo estiver rodando sob uma versão de depuração (DBG) do Windows NT, tal como aquela distribuída no CD-ROM SDK, **GlobalFree** e **LocalFree** colocarão um breakpoint imediatamente antes de liberar um objeto bloqueado. Isso permitirá que um programa confira outra vez o comportamento desejado. Digitar G ao usar o depurador nessa situação permite que a operação de liberação aconteça.*

1366 USANDO GLOBALLOCK E GLOBALHANDLE

Seus programas podem usar as funções *GlobalAlloc* e *GlobalReAlloc* para alocar memória a partir do heap global. No entanto, como você viu, ambas as funções de alocação retornam um indicativo para a memória alocada. Por outro lado, você irá querer que a maior parte de seus programas use essa memória com um ponteiro. Você pode usar as funções *GlobalLock* e *GlobalHandle* para facilmente converter memória alocada em um ponteiro e de volta para o indicativo outra vez. A função *GlobalLock* bloqueia um objeto de memória global e retorna um ponteiro para o primeiro byte do bloco de memória do objeto. Seus programas não podem mover ou descartar o bloco de memória associado com um objeto de memória bloqueado. Para os objetos de memória que você alocar com o sinalizador *GMEM_MOVEABLE*, a função incrementará o contador de bloqueio associado com o objeto de memória. Você usará a função *GlobalLock*, como segue:

```
LPVOID GlobalLock(
   HGLOBAL hMem     // endereço do objeto de memória global
);
```

Na função *GlobalLock*, o parâmetro *hMem* identifica o objeto de memória global. Ou a função *GlobalAlloc* ou *GlobalReAlloc* retornarão esse indicativo. Se a função *GlobalLock* for bem-sucedida, o valor de retorno será um ponteiro para o primeiro byte do bloco de memória. Se a função *GlobalLock* falhar, o valor de retorno será *NULL*.

As estruturas de dados internas para cada objeto de memória incluem um contador de bloqueio que é inicialmente zero. Para os objetos de memória móveis, *GlobalLock* incrementa o contador, e a função *GlobalUnlock* decrementa o contador. Para cada chamada que um processo faz à *GlobalLock* para um objeto, o processo precisa eventualmente chamar *GlobalUnlock*. Seu programa não pode mover ou descartar a memória bloqueada, a não ser que o programa realoque o objeto de memória usando a função *GlobalReAlloc*. Um bloco de memória de objeto de memória bloqueada permanece bloqueado até que seu programa decremente o contador de bloqueio do bloco de memória para zero, instante em que seu programa pode mover ou descartar a memória.

Os objetos de memória alocados anteriormente com o sinalizador *GMEM_FIXED* sempre têm um contador de bloqueio de zero. Para esses objetos, o valor do ponteiro retornado é igual ao valor do indicativo especificado. Se seu programa tiver descartado anteriormente o bloco de memória especificado ou se o bloco de memória tiver um tamanho de zero bytes, a função *GlobalLock* retornará *NULL*. Os objetos descartados sempre têm um contador de bloqueio de zero. Você geralmente implementará a função *GlobalLock* com código similar ao mostrado no fragmento a seguir:

```
HGLOBAL hMem = GlobalAlloc(GHND, 27);
LPSTR pCur;

if (hMem && (pMem = (LPTSTR)GlobalLock(hMem)) != NULL)
```

O código precedente aloca um indicativo para 27 bytes de memória (a variável *hMem*), depois bloqueia essa memória no ponteiro *pMem*, que é simultaneamente convertido como um tipo string.

Como você aprendeu, seus programas frequentemente usarão *GlobalLock* para converter os indicativos de memória em ponteiros. Você geralmente usará a função *GlobalHandle* para converter um ponteiro de volta para um indicativo de memória. A função *GlobalHandle* lê o indicativo associado com o ponteiro especificado para um bloco de memória global. A razão mais comum para converter um ponteiro de volta a um indicativo é preparar para um comando *GlobalFree*. Você usará a função *GlobalHandle* dentro de seus programas, como segue:

```
HGLOBAL GlobalHandle(
  LPCVOID pMem   // ponteiro para o bloco de memória global
);
```

O parâmetro *pMem* é um ponteiro para o primeiro byte do bloco de memória global. A função *GlobalLock* retorna esse ponteiro. Se a função *GlobalHandle* for bem-sucedida, o valor de retorno será o indicativo do objeto de memória global especificado. Se a função *GlobalHandle* falhar, o valor de retorno será *NULL*.

Quando a função *GlobalAlloc* aloca um objeto de memória com o sinalizador *GMEM_MOVEABLE*, ela retorna o indicativo do objeto. A função *GlobalLock* converte esse indicativo em um ponteiro para o bloco de memória, e *GlobalHandle* converte o ponteiro de volta em um indicativo. Você geralmente implementará *GlobalHandle* dentro de seu código, como mostrado aqui:

```
HGLOBAL  hMem = GlobalHandle(pMem);

GlobalUnlock(hMem);
pMem = NULL;
hMem = GlobalReAlloc(hMem, (26*2)+1, GMEM_MOVEABLE);
```

Neste caso em particular, o programa cria o indicativo, depois o desbloqueia, liberando a memória. O programa então realoca a memória, conforme é necessário. O código posterior dentro de seus programas provavelmente converterá o indicativo de volta em um ponteiro.

VERIFICANDO A MEMÓRIA DO COMPUTADOR 1367

O Windows mantém informações sobre a memória física e a memória virtual do seu computador. Normalmente, seus programas irão requerer informações sobre a quantidade de memória livre disponível, que seus programas poderão acessar. Seus programas podem usar a função *GlobalMemoryStatus* para descobrir o estado atual da memória do seu computador. A função *GlobalMemoryStatus* retorna informações sobre memória física e virtual. Você usará a função *GlobalMemoryStatus* dentro de seus programas, como mostra o protótipo a seguir:

```
VOID GlobalMemoryStatus(
  LPMEMORYSTATUS lpBuffer     // ponteiro para a
```

```
                                    // estrutura de status da memória
);
```

O parâmetro *lpBuffer* aponta para uma estrutura *LPMEMORYSTATUS* na qual *GlobalMemoryStatus* retorna informações sobre a disponibilidade atual de memória. Antes de chamar a função *GlobalMemoryStatus*, o processo chamador deve definir o membro *dwLength* dessa estrutura. A estrutura *MEMORYSTATUS* contém informações sobre a disponibilidade atual de memória. A API do Windows define a estrutura *MEMORYSTATUS* como segue:

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;        // sizeof(MEMORYSTATUS)
    DWORD dwMemoryLoad;    // porcentagem de memória em uso
    DWORD dwTotalPhys;     // byte da memória física
    DWORD dwAvailPhys;     // bytes da memória física livres
    DWORD dwTotalPageFile; // bytes do arquivo de paginação
    DWORD dwTotalVirtual;  // bytes do espaço de endereço do usuário
    DWORD dwAvailVirtual;  // bytes livres do usuário
} MEMORYSTATUS;
```

Como você pode ver, a estrutura *MEMORYSTATUS* armazena informações significativas sobre a memória disponível atual do computador. A Tabela 1367 explica os membros da estrutura *MEMORYSTATUS*.

Tabela 1367 Os membros da estrutura ***MEMORYSTATUS***.

Membro	Descrição
dwLength	Indica o tamanho da estrutura. O processo chamador deve definir esse membro antes de chamar *GlobalMemoryStatus*.
dwMemoryLoad	Especifica um número entre 0 e 100 que dá uma idéia geral da utilização atual da memória, na qual 0 indica nenhum uso de memória e 100 indica uso total.
dwTotalPhys	Indica o número total de bytes da memória física.
dwAvailPhys	Indica o número de bytes da memória física disponível.
dwTotalPageFile	Indica o número total de bytes que todos os programas podem armazenar em um arquivo de paginação. Observe que esse número não representa o tamanho físico real do arquivo de paginação no disco.
dwAvailPageFile	Indica o número de bytes disponíveis no arquivo de paginação.
dwTotalVirtual	Indica o número total de bytes que o Windows pode descrever na porção do modo do usuário do espaço de endereço virtual do processo chamador.
dwAvailVirtual	Indica o número de bytes de memória não-reservados e não-comprometidos na porção do modo do usuário do espaço do endereço virtual do processo chamador.

Um aplicativo pode usar a função *GlobalMemoryStatus* para determinar quanta memória ela poderá alocar sem causar impacto severamente nos outros aplicativos. As informações que a função *GlobalMemoryStatus* retorna é volátil, e não há garantia de que duas chamadas seqüenciais à função *GlobalMemoryStatus* retornarão as mesmas informações. Para compreender melhor o processamento que a função *GlobalMemoryStatus* executa, considere o programa *Global_Mem_Status.cpp*, contido no CD-ROM que acompanha este livro. O programa *Global_Mem_Status.cpp* verifica o estado atual da memória e exibe essa quantidade de memória na janela do programa.

1368 CRIANDO UM HEAP DENTRO DE UM PROCESSO

Em dicas anteriores, você aprendeu como alocar memória para seus programas a partir do heap global. No entanto, como também aprendeu, seus programas alocarão blocos de memória menores a partir de um heap local (privado). A função *HeapCreate* cria um objeto heap que o processo chamador pode usar. A função reserva um bloco contíguo no espaço de endereço virtual do processo e aloca armazenagem virtual para um porção inicial especificada do bloco reservado. Você usará a função *HeapCreate* dentro de seus programas, como mostra o protótipo a seguir:

```
HANDLE HeapCreate(
  DWORD flOptions,      // sinalizador de alocação do heap
  DWORD dwInitialSize,  // tamanho inicial do heap
  DWORD dwMaximunSize   // tamanho máximo do heap
);
```

O parâmetro *flOptions* especifica atributos adicionais (sinalizadores) para o novo heap. Esses sinalizadores afetarão o acesso subseqüente ao novo heap por meio de chamada às funções do heap (*HeapAlloc, HeapFree, HeapReAlloc* e *HeapSize*). Você pode especificar um ou mais dos sinalizadores listados na Tabela 1368.

Tabela 1368 Os sinalizadores possíveis para o parâmetro ***flOptions***.

Sinalizador	Descrição
HEAP_GENERATE_EXCEPTIONS	Especifica que o sistema gerará uma exceção para indicar uma falha na função, tal como uma condição de falta de memória, em vez de retornar *NULL*.
HEAP_NO_SERIALIZE	Especifica que o heap não usará exclusão mútua quando as funções do heap alocarem e liberarem memória a partir do heap. O padrão, que ocorre quando você não especifica o sinalizador *HEAP_NO_SERIALIZE*, é serializar o acesso ao heap. A serialização do acesso ao heap permite que dois ou mais encadeamentos aloquem e liberem memória a partir do heap.

O parâmetro *dwInitialSize* especifica o tamanho inicial, em bytes, do heap. O valor do parâmetro *dwInitialSize* determina a quantidade inicial de armazenagem física que *HeapCreate* aloca para o heap. *HeapCreate* arredonda o valor até o próximo limite de página. Para determinar o tamanho de uma página no computador hospedeiro, use a função *GetSystemInfo*. Se o parâmetro *dwMaximumSize* for um valor diferente de zero, ele especificará o tamanho máximo, em bytes, do heap. A função *HeapCreate* arredonda *dwMaximumSize* até o próximo limite de página, e, depois, reserva um bloco desse tamanho no espaço de endereço virtual do processo para o heap. Se as funções *HeapAlloc* ou *HeapReAlloc* fizerem solicitações de alocação que excederem a quantidade inicial de armazenagem física que *dwInitialSize* especifica, o sistema alocará páginas adicionais de armazenagem física para o heap, até o tamanho máximo do heap.

Além disso, se *dwMaximumSize* for diferente de zero, o heap não poderá crescer, e uma limitação absoluta surgirá: o tamanho máximo de um bloco de memória no heap será um bit menor que *0x0007FFF8* bytes (o tamanho do espaço de endereço privado do processo). As solicitações para alocar blocos maiores falharão, mesmo se o tamanho máximo do heap for grande o suficiente para conter o bloco. Se *dwMaximumSize* for zero, ele especificará que o heap pode crescer. Somente a memória disponível limita o tamanho do heap. As solicitações para alocar blocos maiores que *0x0007FFF8* bytes não falham automaticamente; o sistema chama *VirtualAlloc* para obter a memória necessária para esses blocos grandes. Os aplicativos que precisam alocar blocos grandes de memória devem definir *dwMaximumSize* como zero.

Se a função for bem-sucedida, o valor de retorno será um indicativo do heap recém-criado. Se a função falhar, o valor de retorno será *NULL*. Para obter informações estendidas de erro, você precisa chamar *GetLastError*.

A função *HeapCreate* cria um objeto de heap privado a partir do qual o processo chamador pode usar a função *HeapAlloc* para alocar blocos de memória. O tamanho inicial determina o número de páginas comprometidas que *HeapCreate* aloca inicialmente para o heap. O tamanho máximo determina o número total de páginas reservadas. Essas páginas comprometidas e as páginas reservadas juntas criam um bloco contíguo no espaço de endereço virtual do processo no qual o heap pode crescer. Se *HeapAlloc* fizer solicitações que excedem o tamanho atual das páginas comprometidas, o Windows automaticamente comprometerá as páginas adicionais a partir das páginas reservadas, assumindo que a armazenagem física está disponível.

Somente o processo que criou o objeto de heap privado pode acessar essa memória armazenada dentro do heap privado. Se uma biblioteca de ligações dinâmicas (DLL) cria um heap privado, ela cria o heap no espaço de endereço do processo que o chamou. Além disso, o heap é acessível somente a esse processo. O sistema usa memória a partir do heap privado para armazenar estruturas de apoio, de modo que nem todo o tamanho es-

pecificado do heap está disponível para o processo. Por exemplo, se a função *HeapAlloc* solicitar 64Kb de um heap com um tamanho máximo de 64Kb, a solicitação poderá falhar devido à sobrecarga do sistema.

Se você não especificar o sinalizador *HEAP_NO_SERIALIZE* (o padrão simples), o heap serializará o acesso dentro do processo chamador. A serialização garantirá exclusão mútua quando dois ou mais encadeamentos tentarem alocar simultaneamente ou liberar blocos a partir do mesmo heap. Há um pequeno custo no desempenho para a serialização (isto é, o Windows requer tempo de processamento adicional), mas seus programas precisam *usar a serialização sempre que múltiplos encadeamentos alocarem e liberarem memória a partir do mesmo heap.*

Definir o sinalizador *HEAP_NO_SERIALIZE* elimina a exclusão mútua no heap. Sem a serialização, dois ou mais encadeamentos que usam o mesmo indicativo de heap poderão tentar alocar ou liberar a memória ao mesmo tempo, o que provavelmente causaria corrupção no heap. Portanto, você pode seguramente usar o sinalizador *HEAP_NO_SERIALIZE* apenas nas seguintes situações:

- O processo tem somente um encadeamento.
- O processo tem múltiplos encadeamentos, mas somente um encadeamento chama as funções de heap para um heap específico.
- O processo tem múltiplos encadeamentos, e o aplicativo fornece seu próprio mecanismo para a exclusão mútua para um heap específico.

1369 Usando as Funções do Heap para Gerenciar a Memória Específica do Processo

Como você aprendeu, seus programas devem alocar pequenas quantidades de memória a partir do heap que um processo específico requer. Seus programas tipicamente usarão a função *HeapAlloc* para alocar tal memória. A função *HeapAlloc* aloca um bloco de memória a partir de um heap. A memória que você alocar com *HeapAlloc* não será móvel. Você usará a função *HeapAlloc* dentro de seus programas, como segue:

```
LPVOID HeapAlloc(
   HANDLE hHeap,     // indicativo para o bloco de heap privado
   DWORD dwFlags,    // sinalizadores de controle de alocação do heap
   DWORD dwBytes     // número de bytes a alocar
);
```

O parâmetro *hHeap* especifica o heap a partir do qual *HeapAlloc* alocará a memória. O parâmetro *hHeap* é um indicativo que a função *HeapCreate* ou *GetProcessHeap* retorna. O parâmetro *dwFlags* especifica vários aspectos controláveis da alocação do heap. Especificar qualquer um desses sinalizadores anulará o sinalizador correspondente que você especificou quando criou o heap com *HeapCreate*. Você pode especificar um ou mais dos sinalizadores para o parâmetro *dwFlags*, detalhados na Tabela 1369.1.

Tabela 1369.1 Os sinalizadores para o parâmetro ***dwFlags***.

Sinalizador	Significado
HEAP_GENERATE_EXCEPTIONS	Especifica que o sistema operacional gerará uma exceção para indicar uma falha da função, tal como uma condição de falta de memória, em vez de retornar *NULL*.
HEAP_NO_SERIALIZE	Especifica que o heap não usará a exclusão mútua enquanto a função *HeapCreate* estiver acessando o heap.
HEAP_ZERO_MEMORY	Especifica que o Windows inicializará a memória alocada com zero.

Finalmente, o parâmetro dwBytes especifica o número de bytes que HeapAlloc alocará. Se o parâmetro hHeap especificar um heap não-expandível, *dwBytes* precisará ser menor que *0x7FFF8*. Você chamará a função *HeapCreate* com um valor diferente de zero para criar um heap não-expandível. Se a função for bem-sucedida, o valor de retorno será um ponteiro para o bloco de memória alocada. Se a função falhar, e você tiver especificado *HEAP_GENERATE_EXCEPTIONS*, a função poderá gerar uma exceção com os valores de erro listados na Tabela 1369.2.

Tabela 1369.2 Os valores de erro de uma má alocação de heap.

Valor	Significado
STATUS_NO_MEMORY	A tentativa de alocação falhou por falta de memória disponível ou corrupção do heap.
STATUS_ACCESS_VIOLATION	A tentativa de alocação falhou por causa de corrupção no heap ou parâmetros incorretos da função.

Observe que a corrupção no heap pode levar a ambas exceções; depende da natureza da corrupção. Se *HeapAlloc* for bem-sucedida, ela alocará pelo menos a quantidade de memória que o programa chamador solicitou. Se a quantidade real que *HeapAlloc* aloca for maior que a quantidade que o programa chamador solicita, o processo poderá usar a quantidade toda. Você pode usar a função *HeapSize* para determinar o tamanho real do bloco alocado.

Para liberar um bloco de memória que *HeapAlloc* tiver alocado, use a função *HeapFree*. A memória que *HeapAlloc* alocou não é móvel. Como a memória não é móvel, é possível que o heap se torne fragmentado. Observe que se você não especificar *HEAP_ZERO_MEMORY*, o Windows não inicializará a memória alocada com zero.

Para compreender melhor o processamento que *HeapAlloc* executa, considere o programa *Heap_Strings.cpp*, contido no CD-ROM que acompanha este livro. O programa *Heap_Strings.cpp* cria um heap, depois usa *HeapAlloc* para alocar memória a partir desse heap, que o programa trata como uma matriz dinâmica de strings. Quando o usuário seleciona o item de menu *Alocar!*, o programa aloca memória a partir do heap para armazenar uma nova string bem como declarar o ponteiro para a memória que a alocação anterior acrescentou na matriz. Se não houver espaço livre na matriz, o programa usará *HeapReAlloc* para expandir a matriz. Quando o usuário selecionar o item de menu *Liberar!*, o programa liberará a memória para a última string alocada. Quando o programa detectar que o usuário liberou memória suficiente para deixar uma quantidade significativa de espaço não-usado, ele realocará o heap para compactar a matriz. Além disso, toda vez que o programa *Heap_Strings.cpp* realocar o heap, ele usará a função *HeapCompact* para compactar o heap.

VERIFICANDO O TAMANHO DA MEMÓRIA ALOCADA A PARTIR DE UM HEAP 1370

Seus programas Windows freqüentemente alocam pequenas quantidades de memória local a partir de um heap privado. Em dicas anteriores, você criou um heap e alocou memória nele. Seus programas também podem usar funções, tais como *HeapReAlloc*, para realocar espaço a partir de um heap, e *HeapFree*, para liberar a memória que você aloca em um heap. Adicionalmente, seus programas sempre devem usar a função *HeapDestroy* para destruir os heaps privados que criarem. Freqüentemente, no entanto, durante a execução do seu programa, você irá querer verificar o tamanho de uma alocação que fez no heap. Seus programas podem usar a função *HeapSize* para verificar o tamanho de um bloco de memória alocado no heap. A função *HeapSize* retorna o tamanho, em bytes, de um bloco de memória que a função *HeapAlloc* ou *HeapReAlloc* alocou em um heap. Você usará a função *HeapSize* dentro de seus programas, como mostrado no protótipo a seguir:

```
DWORD HeapSize(
   HANDLE hHeap,      // indicativo do heap
   DWORD dwFlags,     // sinalizadores de controle do tamanho
   LPCVOID lpMem      // ponteiro para memória para o retorno do tamanho
```

O parâmetro *hHeap* especifica o heap em que o bloco de memória reside. A função *HeapCreate* ou *GetProcessHeap* retornará esse indicativo. O parâmetro *dwFlags* especifica vários aspectos controláveis de acessar o bloco de memória. Atualmente, você apenas pode especificar o sinalizador *HEAP_NO_SERIALIZE*, no entanto, o Windows reserva todos os outros valores de sinalizadores para uso futuro. Especificar o sinalizador *HEAP_NO_SERIALIZE* anulará o sinalizador correspondente que você especificou no parâmetro *flOptions* quando usou a função *HeapCreate* para criar o heap. Finalmente, o parâmetro *lpMem* aponta para o bloco de memória cujo tamanho a função obterá. Isso é um ponteiro que a função *HeapAlloc* ou *HeapReAlloc* retorna.

Se a função *HeapSize* for bem-sucedida, o valor de retorno será o tamanho, em bytes, do bloco de memória alocado. Se a função *HeapSize* falhar, o valor de retorno será *0xFFFFFFFF*.

Para compreender melhor o processamento que a função *HeapSize* realiza, considere o programa *Heap_Size.cpp* contido no CD-ROM que acompanha este livro. Esse programa cria um heap e aloca um bloco de memória zerada de 20 bytes de comprimento. O programa depois chama a função *HeapSize* para exibir o tamanho do bloco alocado.

1371 ALOCANDO UM BLOCO DE MEMÓRIA VIRTUAL

Como você aprendeu, seus programas Windows geralmente alocarão memória usando um dentre três tipos de alocação — alocação no heap global, alocação no heap privado, ou alocação direta de memória virtual. Usar memória virtual oferece ao seus programas opções adicionais e controle sobre o processo de alocação. No entanto, como com os outros tipos de alocação, você ainda efetuará alocação de memória com uma função *alloc*. A função *VirtualAlloc* difere de outras funções de alocação em que ela reserva ou compromete uma região de páginas no espaço de endereço virtual do processo chamador. O Windows inicializa a memória automaticamente que seus programas alocam usando a função VirtualAlloc para zero. Você usará a função *VirtualAlloc* dentro de seus programas, como segue:

```
LPVOID VirtualAlloc(
   LPVOID lpEnder,     // endereço da região a reservar ou comprometer
   DWORD dwTamanho,    // tamanho da região
   DWORD flTipoAloc,   // tipo da alocação
   DWORD flProtege     // tipo de proteção de acesso
);
```

A função *VirtualAlloc* aceita os parâmetros que a Tabela 1371.1 detalha.

Tabela 1371.1 Os parâmetros para a função ***VirtualAlloc***.

Parâmetro	Descrição
lpEnder	Especifica o endereço inicial da região que você quer alocar. Se seu programa estiver reservando a memória, o Windows arredondará o endereço especificado para o próximo limite de 64Kb. Se seu programa já reservou a memória e agora estiver chamando *VirtualAlloc* para comprometer a memória, o Windows arredondará para o próximo limite de página. Para determinar o tamanho de uma página no computador hospedeiro, use a função *GetSystemInfo*. Se esse parâmetro for *NULL*, o sistema determinará onde alocar a região.
dwTamanho	Especifica o tamanho, em bytes, da região. Se o parâmetro *lpEnder* for *NULL*, o Windows arredondará até o valor *dwTamanho* para o próximo limite de página. Caso contrário, as páginas alocadas incluirão todas as páginas que contêm um ou mais bytes no intervalo de *lpEnder* até (*lpEnder+dwTamanho*) Isso significa que um intervalo de 2 bytes que esteja em um limite de página fará o Windows incluir ambas as páginas na região alocada.
flTipoAloc	Especifica o tipo de alocação. Você pode especificar qualquer combinação dos sinalizadores que a Tabela 1371.2 detalha.
flProtege	Especifica o tipo de proteção de acesso. Se você estiver usando *VirtualAlloc* para reservar as páginas, poderá especificar qualquer um dos sinalizadores que a Tabela 1371.2 detalha, juntamente com os sinalizadores de modificadores da proteção *PAGE_GUARD* e *PAGE_NOCACHE*, conforme desejado.

Como a Tabela 1371.1 explicou, você usa o parâmetro *flTipoAloc* para controlar a alocação que *VirtualAlloc* executa. Você pode especificar qualquer combinação dos sinalizadores detalhados na Tabela 1371.2 para controlar a alocação da memória virtual.

Tabela 1371.2 Os tipos possíveis de alocação para a função *VirtualAlloc*.

Sinalizador	Significado
MEM_COMMIT	Aloca armazenagem física na memória ou no arquivo de paginação do disco para a região de páginas especificadas — em outras palavras, protege parte do espaço de endereço virtual do processo das outras chamadas de alocação dentro do mesmo processo. Tentar reservar uma página já reservada não fará a função falhar. Isso significa que seus programas podem comprometer um intervalo de páginas reservadas ou não-reservadas sem ter que se preocupar com uma falha.
MEM_RESERVE	Reserva um intervalo do espaço de endereço virtual do processo sem alocar qualquer armazenagem física. Quaisquer outras operações de alocação (tais como a função *malloc*, ou *GlobalAlloc*) não podem usar o intervalo reservado até que seu programa libere o intervalo. Seus programas podem comprometer páginas reservadas em chamadas subseqüentes à função *VirtualAlloc*.
MEM_TOP_DOWN	Aloca memória no maior endereço possível.

Devido à natureza da alocação de páginas virtuais, é possível controlar o acesso às páginas virtuais que você compromete com a função *VirtualAlloc*. Como a Tabela 1371.1 observou, você pode especificar um tipo único de segurança de página, junto com os modificadores *PAGE_GUARD* e *PAGE_NOCACHE*. A Tabela 1371.3 lista os sinalizadores de segurança que você pode usar ao alocar as páginas virtuais.

Tabela 1371.3 Os sinalizadores de segurança possíveis para as alocações de página virtual.

Sinalizador	Significado
PAGE_READONLY	Habilita o acesso de leitura apenas à região de página comprometida. Tentar escrever nas páginas de leitura somente resultarão em uma violação de acesso. Se o sistema diferenciar entre o acesso de leitura somente e executar o acesso, tentar executar código na região comprometida resultará em uma violação de acesso.
PAGE_READWRITE	Habilita tanto o acesso de leitura quanto o de gravação na região de página comprometida
PAGE_EXECUTE	Habilita o acesso de execução apenas para a região de página comprometida. Tentar ler ou escrever nas páginas de execução somente resultará em uma violação de acesso.
PAGE_EXECUTE_READ	Habilita o acesso de leitura de execução apenas para a região de página comprometida. Tentar escrever nas páginas de leitura ou de execução somente resultará em uma violação de acesso.
PAGE_EXECUTE_READWRITE	Habilita o acesso de execução, leitura e gravação na região de página comprometida.
PAGE_GUARD	Páginas na região se tornam páginas de guarda. Se um programa tentar ler ou escrever em uma página de guarda, a página de guarda fará o sistema operacional gerar uma exceção *STATUS_GUARD_PAGE* e desativar o status da página de guarda. Portanto, as páginas de guarda atuam como um alarme de acesso de um único disparo

Tabela 1371.3 Os sinalizadores de segurança possíveis para as alocações de página virtual. (*Continuação*)

Sinalizador	Significado
	O sinalizador *PAGE_GUARD* é um modificador de proteção de página. Um aplicativo o usa com um dos outros sinalizadores de proteção de página, com uma exceção: um aplicativo não pode usá-lo com *PAGE_NOACCESS*. Após uma ação de leitura ou uma ação de gravação falhar e levar o sistema operacional a desativar o status da página de guarda, a proteção de página subjacente tomará o controle. Se uma exceção de página de guarda ocorrer durante um serviço do sistema, o serviço tipicamente retornará um indicador de status de falha. A dica a seguir explicará em detalhe as páginas de guarda.
PAGE_NOACCESS	Desabilita todo o acesso à região de página comprometida. Tentar ler, escrever ou executar em páginas de acesso desabilitado resultará em uma exceção de violação de acesso, chamada de falha de proteção geral (GP).
PAGE_NOCACHE	Não permite nenhum cache das páginas anteriormente alocadas com o sinalizador *MEM_COMMIT*. Você deve especificar os atributos de hardware para a memória física como "nenhum cache". A Microsoft não recomenda esse sinalizador para o uso geral. Ele é útil para os controladores de dispositivo (por exemplo, mapear um buffer de imagens de vídeo) que não usam cache. Esse sinalizador é um modificador de proteção de página, válido somente quando você o usa com uma das outras proteções de página que não *PAGE_NOACCESS*

Se a função *VirtualAlloc* for bem-sucedida, o valor de retorno será o endereço-base da região alocada de páginas. Se a função falhar, o valor de retorno será *NULL*.

VirtualAlloc pode efetuar as seguintes operações:

- Comprometer uma região de páginas que uma chamada anterior à função *VirtualAlloc* reservou.
- Reservar uma região de páginas livres.
- Reservar e comprometer uma região de páginas livres.

Você pode usar *VirtualAlloc* para reservar um bloco de páginas e depois fazer chamadas adicionais à *VirtualAlloc* para comprometer páginas individuais a partir do bloco reservado. Reservar um bloco de páginas, habilita um processo a reservar um intervalo de seu espaço de endereço virtual sem consumir armazenagem física até que você precise dele.

Cada página no espaço de endereço virtual do processo está em um dos três estados que a Tabela 1371.4 detalha

Tabela 1371.4 Os estados possíveis para a memória virtual.

Estado	Significado
Livre	O processo não comprometeu ou reservou esta página, e, assim, ela não está acessível para o processo. *VirtualAlloc* pode reservar, ou simultaneamente reservar ou comprometer uma página livre.
Reservado	Outras funções de alocação não podem usar o intervalo de endereços, mas o processo não pode acessar a página e o Windows não associa nenhuma armazenagem física com a página. *VirtualAlloc* pode comprometer uma página reservada, mas ela não pode reservá-la uma segunda vez. A função *VirtualFree* pode liberar uma página reservada, fazendo dela uma página livre.

Tabela 1371.4 Os estados possíveis para a memória virtual. (*Continuação*)

Estado	Significado
Comprometida	O Windows alocou armazenagem para a página, e um código de proteção controla o acesso. O sistema inicializa e carrega cada páginas comprometida na memória física somente na primeira tentativa para ler ou escrever naquela página. Quando o processo terminar, o sistema liberará a armazenagem para as páginas comprometidas. VirtualAlloc pode comprometer uma página já comprometida. Isso significa que você poderá comprometer um intervalo de páginas, independentemente se você já as comprometeu ou não, e a função não falhará. VirtualFree pode descomprometer uma página comprometida, liberando a armazenagem de memória, ou pode descomprometer e liberar uma página comprometida.

Se o parâmetro *lpEnder* não for *NULL*, a função usará os parâmetros *lpEnder* e *dwTamanho* para calcular a região de páginas que *VirtualAlloc* alocará. O estado atual do intervalo inteiro de páginas precisa ser compatível com o tipo de alocação que o parâmetro *flTipoAloc* especifica. Caso contrário, a função falhará e *Virtual_Allocate* não alocará nenhuma página. Esse requisito de compatibilidade não impede o comprometimento de uma página já comprometida (veja a Tabela 1371.4).

Para compreender melhor o processamento que *VirtualAlloc* realiza, considere o programa *VirtualAllocate*, contido no CD-ROM que acompanha este livro. Esse programa reserva 1Mb de memória virtual quando o envia à mensagem *WM_CREATE*. Quando o usuário seleciona o item de menu *Testar!*, o programa compromete e usa 70Kb de memória virtual. Primeiro, o programa coloca valores em cada bloco de Kb da memória alocada. Segundo, o programa altera o acesso do bloco de memória comprometido inteiro para leitura somente. Terceiro, o programa acessa um valor na memória e o exibe dentro de uma caixa de mensagem. Finalmente, o programa tenta definir um valor na memória, o que causa uma falha de proteção. O programa *Virtual_Allocate* usa um bloco *try-catch* para pegar a falha de proteção.

COMPREENDENDO AS PÁGINAS DE GUARDA 1372

Como visto na dica anterior, um aplicativo define um sinalizador modificador de página de proteção *PAGE_GUARD* para estabelecer uma *página de guarda*. Você pode especificar esse sinalizador, juntamente com outros sinalizadores de proteção de página, nas funções *VirtualAlloc*, *VirtualProtect* e *VirtualProtectEx*. Você pode usar o sinalizador *PAGE_GUARD* com qualquer outro sinalizador de proteção de página, com exceção de *NO_ACCESS*.

Se um programa tentar acessar um endereço dentro de uma página de guarda, o sistema operacional gerará uma exceção *STATUS_GUARD_PAGE* (*0x80000001*). O sistema operacional também limpará o sinalizador *PAGE_GUARD*, removendo o status da página de guarda da página de memória. O sistema não parará na próxima tentativa de acessar a página de memória com uma exceção *STATUS_GUARD_PAGE*.

Se uma exceção de página de guarda ocorrer durante um serviço do sistema, o serviço falhará e tipicamente retornará algum indicador de status de falha. Como o sistema também remove o status da página de guarda relevante da página de memória, a próxima chamada do mesmo serviço do sistema não falhará devido a uma exceção *STATUS_GUARD_PAGE* (a não ser, é claro, que alguém reestabeleça a página de guarda).

Portanto, uma página de guarda fornece um alarme de um único disparo para o acesso à página de memória. Isso pode ser útil para um aplicativo que precise monitorar o crescimento de grandes estruturas dinâmicas de dados. Por exemplo, alguns sistemas operacionais usam páginas de guarda para implementar a verificação automática da pilha.

O CD-ROM que acompanha este livro inclui o programa *Guard_Page.cpp*, que ilustra o comportamento da proteção de página de guarda e como ela pode usar um serviço do sistema para falhar (o programa gera saída em uma janela do DOS). Quando você compilar e executar o programa *Guard_Page.cpp*, ele exibirá a seguinte saída na sua tela:

```
Committed 512 bytes at address 003F0000
Cannot lock at 003F0000, error = 0x80000001
2nd Lock Achieved at 003F0000
C:\>
```

Observe que a primeira tentativa de bloquear o bloco de memória falha, gerando uma exceção *STATUS_GUARD_PAGE*. A segunda tentativa passa porque a primeira desligou a proteção de página de guarda do bloco de memória.

1373 COMPREENDENDO MELHOR OS BLOCOS DE MEMÓRIA VIRTUAIS

Como você aprendeu, a memória virtual lhe oferece meios adicionais e eficientes de alocar grandes blocos de memória. No entanto, visualizar os benefícios programáticos da memória virtual geralmente é difícil. Um dos modos mais fáceis de ver os benefícios da memória virtual é considerar uma grande matriz de uma estrutura complexa, talvez uma estrutura do tipo planilha. Se você fosse criar uma matriz bidimensional, a definição seria similar à seguinte:

```
Celula MATRIZGRANDE[200][256];
```

Se o tamanho da estrutura *Celula* fosse 128 bytes, ela requereria 6.533.600 (200 x 256 x 128) bytes de memória física. Claramente, essa é uma quantidade significativa de memória para alocar para uma planilha — particularmente quando se considera que a maioria das planilhas não usa todas essas células.

Alternativamente, você poderia usar uma lista ligada para criar a estrutura do tipo planilha. Com a abordagem da lista ligada, você somente precisa criar estruturas *Celulas* para as células na planilha que realmente contêm dados. Como a maioria das células em uma planilha não é usada, o método da lista ligada poupa uma quantidade de memória considerável. No entanto, usar uma lista ligada torna muito mais difícil obter o conteúdo de uma célula. Por exemplo, caso seu programa requisesse o conteúdo da célula na linha 5, coluna 10, ele precisaria percorrer a lista ligada para encontrar a célula — reduzindo desse modo a velocidade do processamento.

A memória virtual oferece um meio-termo entre a declaração de uma matriz bidimensional e a implementação de listas ligadas complexas. Com memória virtual, você obtém o acesso rápido e fácil da técnica de matriz e a capacidade superior de armazenamento das listas ligadas. Para compreender melhor os benefícios da técnica da lista ligada, seu programa poderia fazer o seguinte:

1. Reservar uma região grande o bastante para conter a matriz inteira de estruturas *Celula*. Como você aprendeu, reservar uma região não usa memória física.
2. Localizar o endereço de memória na região reservada onde a estrutura *Celula* deve ficar quando o usuário insere dados em uma célula
3. Comprometer apenas memória física suficiente para o endereço de memória que você localizou no Passo 2 para uma estrutura *Celula*.
4. Definir os membros da nova estrutura *Celula*.

Após você ter mapeado a memória física na posição adequada, seu programa poderá acessar a memória sem criar uma violação de acesso. Claramente, a técnica de memória virtual é uma melhoria significativa em relação às outras técnicas, pois o programa comprometerá memória física somente se o usuário inserir dados nas células da planilha. Como a maioria das células da planilha está vazia, o programa não usará a maior parte da região reservada. Você pode usar a alocação de memória para proporcionar a seus programas acesso rápido a um grande número de membros sem muitos dos sacrifícios que fez com os modelos de memória anteriores.

1374 LIBERANDO MEMÓRIA VIRTUAL

Seus programas poderão usar a função *VirtualAlloc* para reservar ou comprometer páginas de memória virtual. Após você ter reservado ou comprometido páginas de memória virtual, seus programas poderão usar a função *VirtualFree* para liberar ou descomprometer essas páginas. A função *VirtualFree* libera ou descomprome te (ou ambos) uma região de páginas dentro do espaço de endereço virtual do processo chamador. Você usará a função *VirtualAlloc* como segue:

```
BOOL VirtualFree(
   LPVOID  lpEnder,    // endereço da região de páginas comprometidas
   DWORD dwTamanho,    // tamanho da região
   DWORD dwTipoLiber   // tipo da operação de liberação
);
```

O parâmetro *lpEnder* aponta para o endereço-base da regiao de páginas que *VirtualFree* deverá liberar. Se o parâmetro *dwTipoLiber* incluir o sinalizador *MEM_RELEASE*, o parâmetro *lpEnder* precisará ser o endereço-base que a função *VirtualAlloc* retornou quando você reservou uma região de páginas. O parâmetro *dwTamanho* especifica o tamanho, em bytes, da região que *VirtualFree* deverá liberar. Se o parâmetro *dwTipoLiber* incluir o sinalizador *MEM_RELEASE*, o parâmetro *dwTamanho* precisará ser zero. Caso contrário, a região de páginas afetadas incluirá todas as páginas que contêm um ou mais bytes no intervalo do parâmetro *lpEnder* até (*lpEnder+dwTamanho*). Isso significa que um intervalo de 2 bytes que esteja perto de um limite de página fará *VirtualFree* liberar ambas as páginas. O parâmetro *dwTipoLiber* especifica o tipo de operação de liberação. Quando você chamar a função *VirtualFree*, usará um, mas não ambos, dos sinalizadores da Tabela 1374.

Tabela 1374 Os sinalizadores para o parâmetro ***dwTipoLiber***.

Sinalizador	Significado
MEM_DECOMMIT	Descomprometer a região especificada de páginas comprometidas. Tentar descomprometer uma página não-comprometida não faz a função falhar. Isso significa que seus programas podem descomprometer um intervalo de páginas comprometidas ou não-comprometidas sem ter que se preocupar com uma falha.
MEM_RELEASE	Libera a região específica das páginas reservadas. Se seu programa especificar esse sinalizador, o parâmetro *dwTamanho* precisará ser zero ou a função falhará.

Seus programas podem usar a função *VirtualFree* para efetuar uma das seguintes operações:

- Descomprometer uma região de páginas comprometidas ou não-comprometidas.
- Liberar uma região de páginas reservadas.
- Descomprometer e liberar uma região de páginas comprometidas ou não-comprometidas.

Para liberar uma região de páginas, um intervalo inteiro de páginas precisa estar no mesmo estado (todas reservadas ou todas comprometidas), e a função *VirtualFree* precisa liberar toda a região originalmente reservada ao mesmo tempo. Se você tiver usado anteriormente *VirtualAlloc* para comprometer somente algumas das páginas na região reservada original, precisará primeiro chamar *VirtualFree* para descomprometer as páginas comprometidas e depois chamar *VirtualFree* novamente para liberar o bloco inteiro.

As páginas que seus programas usaram *VirtualFree* para liberar são páginas livres disponíveis para operações subseqüentes de alocação. Tentar ler ou escrever em um página liberada resultará em uma exceção de violação de acesso. *VirtualFree* pode descomprometer uma página não-comprometida, o que significa que *VirtualFree* pode descomprometer um intervalo de páginas comprometidas ou não-comprometidas sem ter que se preocupar com uma falha. Descomprometer uma página libera sua armazenagem física, ou na memória ou no arquivo de paginação no disco. Se seu programa descomprometer uma página mas não liberar a página, o estado dessa página mudará para reservado, e uma chamada subseqüente a *VirtualAlloc* poderá comprometer a página novamente. Tentar ler ou escrever em uma página reservada resulta em uma exceção de violação de acesso.

O estado atual do intervalo inteiro de páginas precisa ser compatível com o tipo de operação livre que o parâmetro *dwTipoLivre* especifica. Caso contrário, a função *VirtualFree* falhará e não liberará ou descompromissará quaisquer páginas.

GERENCIANDO AS PÁGINAS DE MEMÓRIA VIRTUAL 1375

Como você aprendeu, seus programas usarão páginas de memória virtual, tanto para expandir o acesso de seu programa quanto ajudar seus programas a alocar grandes blocos de memória dentro do espaço de memória virtual. Geralmente, você irá requerer informações sobre um intervalo de páginas dentro do espaço de endereço virtual de um processo. A função *VirtualQuery* fornece informações sobre um intervalo de páginas no espaço de endereço virtual do processo chamador, como mostrado aqui:

```
DWORD VirtualQuery(
  LPCVOID lpEnder,                          // endereço da região
  PMEMORY_BASIC_INFORMATION lpBuffer,       // endereço do buffer de informações
  DWORD dwTamanho                           // tamanho do buffer
);
```

A função *VirtualQuery* aceita os três parâmetros mostrados na Tabela 1375.

Tabela 1375 Os parâmetros para a função ***VirtualQuery***

Parâmetro	Descrição
lpEnder	Aponta para o endereço-base da região de páginas que *VirtualQuery* consultará. O Windows arredonda este valor para baixo até o próximo limite de página. Para determinar o tamanho de uma página no computador hospedeiro, use a função *GetSystemInfo*.
lpBuffer	Aponta para uma estrutura *MEMORY_BASIC_INFORMATION* na qual *VirtualQuery* retorna informações sobre o intervalo de página especificado.
dwTamanho	Especifica o tamanho, em bytes, do buffer para o qual o parâmetro *lpBuffer* aponta.

VirtualQuery fornece informações sobre uma região de páginas consecutivas começando em um endereço especificado que compartilha os seguintes atributos:

- O estado de todas as páginas é o mesmo com o sinalizador *MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED,* ou *MEM_IMAGE.*
- Se a página inicial não estiver livre, todas as páginas na região serão parte da mesma alocação inicial de página que uma chamada à função *VirtualAlloc* reserva.
- O acesso a todas as páginas é o mesmo com os sinalizadores *PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_ EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD* ou *PAGE_ NOCACHE.*

A função *VirtualAlloc* determina os atributos da primeira página na região, e, depois, percorre páginas subseqüentes até examinar todo o intervalo de página ou até encontrar uma página com um conjunto não-concordante de atributos. *VirtualAlloc* retorna os atributos e o tamanho, em bytes, da região de página com atributos concordantes. Por exemplo, se houver uma região de 40Mb de memória livre e seu programa chamar *VirtualQuery* em uma página que está 10Mb na região, a função obterá um estado de *MEM_FREE* e um tamanho de 30Mb.

A função *VirtualQuery* retorna informações sobre uma região de páginas na memória do processo chamador, e a função *VirtualQueryEx* informa sobre uma região de página na memória de um processo especificado. O CD-ROM que acompanha este livro inclui o programa *Virtual_Query.cpp*, que primeiro aloca um bloco de 70Kb de memória virtual. O programa depois chama a função *VirtualQuery*, que retorna o tamanho da região que a memória realmente ocupa. Observe que o tamanho de uma região é divisível por 4.096 bytes (4Kb), que é o tamanho da página de todos os computadores x86.

1376 COMPREENDENDO MELHOR OS PROCESSOS

Como você aprendeu, um dos aspectos mais poderosos do Windows é seu suporte para a multitarefa, ou a execução de múltiplos processos na memória simultaneamente. Nas trinta dicas a seguir, você aprenderá mais sobre o gerenciamento de processos e de encadeamentos, um dos conhecimentos mais importantes para o programador Windows.

Um *processo* é o objeto que possui todos os recursos de um aplicativo. Um proceso do Windows, por sua vez, pode criar um ou mais encadeamentos Um *encadeamento* é um caminho de execução independente dentro de um processo que compartilha seu espaço de endereço, código e dados globais. Cada encadeamento tem seu próprio conjunto de registradores, sua própria pilha e seus próprios mecanismos de entrada, incluindo uma fila de mensagem privada O Windows 95 e o Windows NT alocam fatias de tempo da CPU encadeamento por encadeamento, e efetuam multitarefa preemptiva (em outras palavras, movem os encadeamentos em frente um do outro) com base na prioridade atribuída a cada encadeamento.

Adicionalmente, um processo incluirá alocações de memória global, páginas virtuais e assim por diante. A Figura 1376.1 mostra um modelo lógico do relacionamento entre encadeamentos e processos.

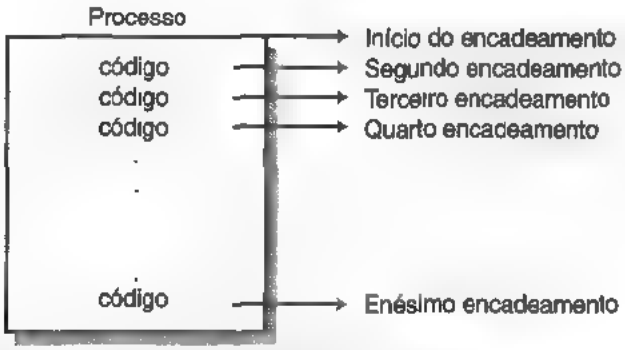

Figura 1376.1 O relacionamento entre os encadeamentos e os processos.

Como você aprendeu, o Windows alocará memória virtual para os processos à medida que eles forem precisando. Portanto, quando você considerar o modelo de memória para um computador que roda múltiplos processos ao mesmo tempo, será importante identificar qual processo estará ativo. Determinar o processo atualmente ativo é importante porque o Windows automaticamente dá prioridade maior à maioria das solicitações da CPU que o processo atualmente ativo executa. A Dica 1400, logo à frente, discutirá o gerenciamento do sistema operacional da prioridade dos encadeamentos em detalhes. Além disso, o Windows tipicamente alocará memória física adicional para agilizar o processamento do processo atualmente ativo. A Figura 1376.2 mostra um exemplo de modelo de memória para três aplicativos simultâneos, com o aplicativo *Prim_Apl* atualmente ativo.

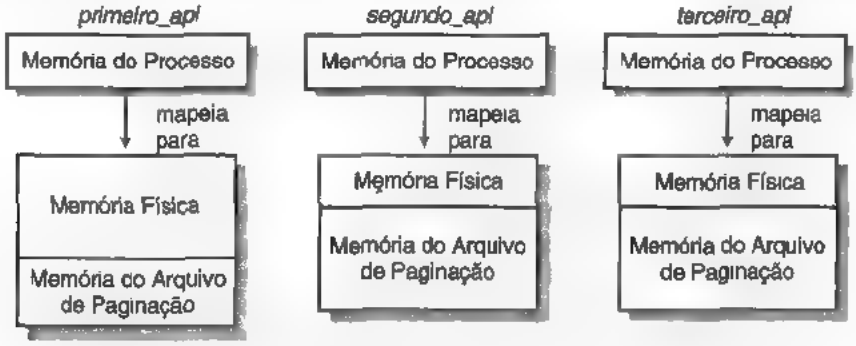

Figura 1376.2 Exemplo de um modelo de memória para três aplicativos em execução simultânea.

No entanto, no caso de o usuário ou o Windows tornar o programa *Segundo_apl* o programa ativo, o Windows realocará memória física de uma maneira que liberará mais espaço para a execução do programa *Segundo_apl*, como mostra a Figura 1376.3.

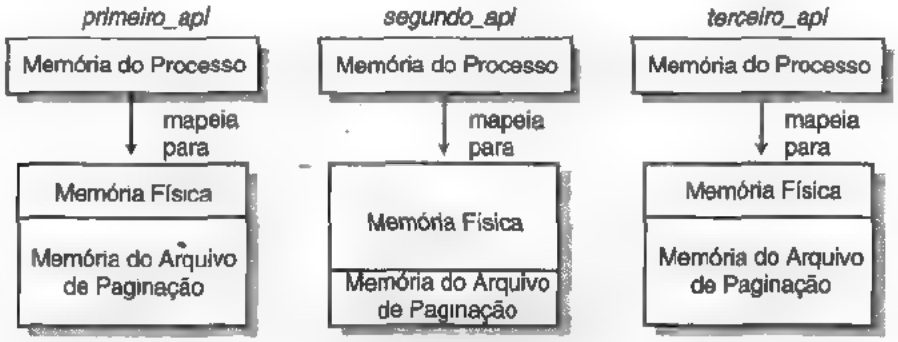

Figura 1376.3 Outro exemplo de modelo de memória para três aplicativos em execução simultânea.

Nas próximas cinco dicas, você aprenderá os fundamentos sobre a criação e o gerenciamento de processos. Em dicas subseqüentes você aprenderá os fundamentos da criação e do gerenciamento dos encadeamentos. Assegure-se de que compreende bem os fundamentos sobre os processos (que são essencialmente repositórios para os encadeamentos) antes de começar a trabalhar com os encadeamentos.

1377 CRIANDO UM PROCESSO

Seus programas geralmente rodarão dentro de um único processo. Esse processo poderá conter um ou mais encadeamentos. A função *CreateProcess* cria um novo processo e seu encadeamento principal (normalmente conhecido como *processo-filho*). O novo processo executará o arquivo executável que você especificar. *CreateProcess* permite que o processo-pai (isto é, o processo que chama a função *CreateProcess*) especifique o ambiente operacional do novo processo, incluindo seu diretório de trabalho, como ele deve aparecer na tela, suas variáveis do ambiente e sua prioridade. O Windows passa a linha de comando e seu conteúdo para o processo-filho. Você usará a função *CreateProcess* como mostrado no protótipo a seguir:

```
BOOL CreateProcess(
   LPCTSTR lpNomeAplic,            // nome do módulo executável
   LPTSTR lpLinhaCmd,              // string da linha de comando
   LPSECURITY_ATTRIBUTES lpAtribProc, // atributos de segurança do processo
   LPSECURITY_ATTRIBUTES lpAtribEncad, // atributos de segurança do
                                       encadeamento
   BOOL bIndicHeranca, // sinalizador do indicativo de herança
   DWORD dwSinalizCriacao,  // sinalizadores de criação
   LPVOID lpAmbiente, // ponteiro para o novo bloco do ambiente
   LPCTSTR lpDiretAtual, // ponteiro para o nome do diretório atual
   LPSTARTUPINFO  lpStartupInfo // ponteiro para STARTUPINFO
   LPPROCESS_INFORMATION lpInfoProcesso  // ponteiro para PROCESS_INFORMATION
);
```

Como você pode ver, *CreateProcess* é uma função que utiliza muitos parâmetros. A Tabela 1377.1 explica os parâmetros para a função *CreateProcess*.

Tabela 1377.1 Os parâmetros para a função ***CreateProcess***

Parâmetros	Descrição
lpNomeAplic	Ponteiro para uma string terminada por *NULL* que especifica o módulo a executar. A string pode especificar o caminho completo e o nome de arquivo do módulo a executar ou a string pode especificar um nome parcial. Quando a string especifica apenas um nome parcial, a função usa a unidade atual e o diretório atual para completar a especificação. O parâmetro *lpNomeAplic* pode ser *NULL*, neste caso o nome do módulo precisa ser o primeiro símbolo delimitado por espaço em branco na string *lpLinhaCmd*. O módulo especificado pode ser um aplicativo baseado no Win32 ou pode ser algum outro tipo de módulo (por exemplo, MS-DOS ou OS/2) caso o computador local tenha o subsistema apropriado disponível.
Nota: Sob o Windows NT, se o módulo executável for um aplicativo de 16 bits, lpNomeAplic deverá ser NULL, e a string para a qual lpLinhaCmd aponta deverá especificar o módulo executável.	
lpLinhaCmd	Ponteiro para uma string terminada por *NULL* que especifica a linha de comando a executar. O parâmetro *lpLinhaCmd* pode ser *NULL*, neste caso a função usa a string para a qual *lpNomeAplic* aponta como a linha de comando. Se tanto *lpNomeAplic* e *lpLinhaCmd* forem não-*NULL*, **lpNomeAplic* especificará o módulo a executar e **lpLinhaCmd* especificará a linha de comando. O novo processo pode usar *GetCommandLine* para obter toda a linha de comando. Adicionalmente, os processos em tempo de execução de C podem usar os argumentos *argc* e *argv*. Se *lpNomeAplic* for *NULL*, o primeiro símbolo delimitado por espaço em branco da linha de comando especificará o nome do módulo. Se o nome de arquivo não contiver uma extensão, o Windows assumirá *EXE*. Se o nome de arquivo terminar com um ponto ( ), sem extensão, ou se o nome de arquivo contiver um caminho, o Windows não anexará *EXE*. Se o nome de arquivo não contiver um caminho de diretório, o Windows pesquisará o arquivo executável na seguinte seqüência:

Tabela 1377.1 Os parâmetros para a função ***CreateProcess*** (*Continuação*)

Parâmetros	Descrição
	1. O diretório a partir de onde o aplicativo foi carregado. 2. O diretório atual para o processo-pai. 3. Windows 95: O diretório do sistema do Windows. Use a função *GetSystemDirectory* para obter o caminho desse diretório. 4. Windows NT: Diretório do sistema Windows de 32 bits. Use a função *GetSystemDirectory* para obter o caminho desse diretório O nome do diretório geralmente é *SYSTEM32*. 5. Windows NT: Diretório do sistema Windows de 16 bits. Não há função do Win32 que obtenha o caminho desse diretório, mas a função o pesquisa. O nome do diretório geralmente é *SYSTEM* 6. Diretório do Windows. Use a função *GetWindowsDirectory* para obter o caminho desse diretório. O nome do diretório geralmente é Windows. 7. Os diretórios que o Windows lista na variável de ambiente *PATH* Se o processo que você quer que *CreateProcess* crie for um aplicativo baseado no MS-DOS ou no Windows, *lpLinhaCmd* deverá ser uma linha de comando completa na qual o primeiro elemento é o nome do aplicativo. Como a função *CreateProcess* também trabalha bem para os aplicativos baseados no Win32, você também deverá definir o parâmetro *lpLinhaCmd* para uma linha de comando completa para os programas Win32.
lpAtribProc	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribProc* for *NULL*, os processos-filho não poderão herdar o indicativo.

Nota: Sob o Windows NT, o membro ***lpDescSeguranca*** *da estrutura define um descritor de segurança para o novo processo. Se* ***lpAtribProc*** *for NULL, o processo receberá um descritor de segurança padrão. Sob o Windows 95, a função* ***CreateProcess*** *ignora o membro* ***lpDescSeguranca*** *da estrutura.*

lpAtribEncad	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribEncad* for *NULL*, os processos-filho não poderão herdar o indicativo.

Nota: Sob o Windows NT, o membro ***lpDescSeguranca*** *da estrutura especifica um descritor de segurança para o encadeamento principal. Se* ***lpAtribEncad*** *for NULL, o encadeamento obterá um descritor de segurança padrão. Sob o Windows 95, a função* ***CreateProcess*** *ignora o membro* ***lpDescSeguranca*** *da estrutura.*

bIndicHeranca	Indica se o novo processo herda os indicativos do processo chamador. Se for *True*, o novo processo herdará cada indicativo de abertura herdável no processo chamador. Os indicativos herdados têm o mesmo valor e privilégios de acesso que os indicativos originais.
dwSinalizCriacao	Especifica sinalizadores adicionais que controlam a classe de prioridade e a criação do processo. Seus programas podem especificar os sinalizadores de criação em qualquer combinação mostrada na Tabela 1377.2 a seguir, exceto como observado. O parâmetro *dwSinalizCriacao* também controla a classe de prioridade do novo processo, que o Windows usa para determinar as prioridades de escalonamento dos encadeamentos do processo.

Tabela 1377.1 Os parâmetros para a função ***CreateProcess***. (*Continuação*)

Parâmetros	Descrição
	Se o Windows não especificar qualquer um dos sinalizadores de classe de prioridade na Tabela 1377.2, a classe de prioridade será o padrão *NORMAL_PRIORITY_CLASS*. No entanto, se a classe de prioridade do processo criador for *IDLE_PRIORITY_CLASS*, a classe de prioridade padrão do processo-filho também será *IDLE_PRIORITY_CLASS*. Seus programas podem especificar qualquer um dos sinalizadores de prioridade detalhados na Tabela 1377.3, logo à frente.
lpAmbiente	Aponta para um bloco do ambiente para o novo processo. Se esse parâmetro for *NULL*, o novo processo usará o ambiente do processo chamador. Um bloco de ambiente consiste de um bloco terminado por *NULL* de strings terminadas por *NULL*. Cada string está na forma *nome=valor*. Como a string usa o sinal de igual como um separador, seu programa não pode usá-lo no nome de uma variável do ambiente. Se um aplicativo fornece um bloco do ambiente, em vez de passar *NULL* para esse parâmetro, o Windows não propagará automaticamente as informações atuais de diretório das unidades do sistema para o novo processo. Um bloco do ambiente pode conter caracteres Unicode ou ANSI. Se o bloco do ambiente para o qual *lpAmbiente* aponta contiver caracteres Unicode, o Windows definirá o sinalizador *CREATE_UNICODE_ENVIRONMENT* no campo *dwSinalizCriacao*. Se o bloco contiver caracteres ANSI, esse sinalizador estará zerado. Observe que dois bytes zero terminam um bloco do ambiente ANSI: um byte zero para a última string e mais um byte zero para finalizar o bloco. Além disso, quatro bytes zero terminam um bloco do ambiente Unicode: dois bytes zero para a última string e mais dois bytes zero para finalizar o bloco.
lpDiretAtual	Aponta para uma string terminada por *NULL* que especifica a unidade e diretório atuais para o processo-filho. A string precisa ser um caminho e um nome de arquivos completos que incluem a letra da unidade (por exemplo, "a:"). Se esse parâmetro for *NULL*, o Windows criará o novo processo com a mesma unidade e diretório atuais que o processo chamador. O Windows fornece essa opção principalmente para interfaces de comando que precisam iniciar um aplicativo e especificar sua unidade e seu diretório inicial de trabalho.
lpInfoInicia	Aponta para uma estrutura *STARTUPINFO* que especifica como deve aparecer a janela principal para um novo processo.
lpInfoProcesso	Aponta para uma estrutura PROCESS_INFORMATION que recebe informações de identificação sobre o novo processo.

Como você viu na Tabela 1377.1, o parâmetro *dwFlags* aceita um ou mais sinalizadores de criação e um sinalizador de classe de prioridade. A Tabela 1377.2 detalha os sinalizadores de criação possíveis.

Tabela 1377.2 Os sinalizadores de criação possíveis para o parâmetro ***dwFlags***.

Sinalizador	Significado
CREATE_DEFAULT_ERROR_MODE	O novo processo não herda o modo de erro do processo chamador. Em vez disso, *CreateProcess* dá ao novo processo o modo de erro padrão atual. Um aplicativo chama *SetErrorMode* para definir o modo de erro padrão atual. Esse sinalizador é particularmente útil para os aplicativos multiencadeados de interface da linha de comando que rodam com os erros desabilitados. O comportamento padrão do processo *Create Process* é o novo processo herdar o modo de erro do chamador. Definir esse finalizador muda o comportamento padrão.

Tabela 1377.2 Os sinalizadores de criação possíveis para o parâmetro ***dwFlags***. (*Continuação*)

Sinalizador	Significado
CREATE_NEW_CONSOLE	O novo processo herda um novo console, em vez de herdar o console do pai. Seu programa não pode usar esse sinalizador com o sinalizador *DETACHED_PROCESS*.
CREATE_NEW_PROCESS_GROUP	O novo processo é o processo-raiz de um novo grupo de processo. O grupo de processo inclui todos os processos que são descendentes desse processo-raiz. O identificador de processo do novo grupo de processo é o mesmo que o identificador que o Windows retorna no parâmetro *lpInfoProcesso*.
CREATE_SEPARATE_WOW_VDM	Windows NT apenas: Este sinalizador é válido somente quando você inicia um aplicativo baseado no Windows de 16 bits. Se você ligar esse sinalizador, o Windows rodará o novo processo em uma Máquina Virtual do DOS (VDM) privada. Por padrão, o Windows roda todos os aplicativos baseados no Windows de 16 bits como encadeamentos em uma única VDM compartilhada. A vantagem de rodar processos em VDMs separadas é que um travamento mata somente uma única VDM; os outros programas que estão em execução em outras VDMs continuarão a funcionar normalmente. Além disso, os aplicativos baseados no Windows de 16 bits que o Windows roda em VDMs separadas têm filas de entrada separadas, o que significa que, se um aplicativo travar momentaneamente, os aplicativos em outras VDMs continuarão a receber entrada.
CREATE_SHARED_WOW_VDM	Windows NT somente: O sinalizador é válido apenas quando você inicia um aplicativo baseado no Windows de 16 bits. Se a opção *DefaultSeparateVDM* na seção Windows do arquivo *WIN.INI* for *True*, esse sinalizador fará a função *CreateProcess* ignorar a opção e rodar o novo processo na VDM compartilhada.
CREATE_SUSPENDED	O Windows cria o encadeamento principal do novo processo em um estado suspenso, e não roda o novo processo até que outro encadeamento (dentro de outro processo) chame a função *ResumeThread*.
CREATE_UNICODE_ENVIRONMENT	Se você definir esse sinalizador, o bloco de ambiente para o qual *lpAmbiente* aponta, usará caracteres Unicode. Se você não definir esse sinalizador, o bloco do ambiente usará caracteres ANSI.
DEBUG_PROCESS	Se você definir esse sinalizador, o Windows tratará o processo chamador como um depurador, e o novo processo será o que o processo chamador estiver depurando. O sistema notifica o depurador de todos os eventos de depuração que ocorrem no processo que você está depurando. Se você criar um processo com esse sinalizador ligado, somente o encadeamento chamador (o encadeamento que chamou *CreateProcess*) poderá chamar a função *WaitForDebugEvent*

Tabela 1377.2 Os sinalizadores de criação possíveis para o parâmetro ***dwFlags***. (*Continuação*)

Sinalizador	Significado
DEBUG_ONLY_THIS_PROCESS	Se você não ligou esse sinalizador e o Windows está atualmente depurando o processo chamador, o novo processo se torna outro processo que o depurador do processo chamador está depurando. Se o processo chamador não for um processo que o Windows está depurando no momento, nenhuma ação relacionada com a depuração ocorrerá como resultado desse sinalizador.
DETACHED_PROCESS	Para os processos de console, o novo processo não tem acesso ao console do processo-pai. O novo processo pode chamar a função *AllocConsole* mais tarde para criar um novo console. Você não pode usar esse sinalizador com *CREATE_NEW_CONSOLE*.

Como você aprendeu, seus programas podem selecionar um ou mais sinalizadores de criação para o parâmetro dwFlags. No entanto, seus programas somente podem especificar um único sinalizador de classe de prioridade para o parâmetro *dwFlags*. O sinalizador de classe de prioridade precisa ser um dos valores especificados na Tabela 1377.3.

Tabela 1377.3 Os sinalizadores de prioridade possíveis para a função ***CreateProcess***.

Sinalizador	***Significado***
HIGH_PRIORITY_CLASS	Indica um processo que executa tarefas de tempo crítico que o Windows precisa executar imediatamente para que o processo rode de forma correta. Os encadeamentos de um processo de classe de alta prioridade fazem a preempção dos encadeamentos dos processos de classe de prioridade normal ou de prioridade ociosa. Um exemplo é a Lista de Tarefas do Windows, que precisa responder rapidamente quando o usuário a chama, independentemente da carga sobre o sistema operacional. Você precisa ser extremamente cauteloso ao usar a classe de alta prioridade, pois um aplicativo dessa classe pode usar praticamente todos os ciclos disponíveis.
IDLE_PRIORITY_CLASS	Indica um processo cujos encadeamentos rodam somente quando o sistema está ocioso, e os encadeamentos de qualquer processo que esteja rodando em uma classe de prioridade mais alta fazem a sua preempção. Um exemplo é um protetor de tela. Os processos-filho herdam a classe de prioridade ociosa.
NORMAL_PRIORITY_CLASS	Indica um processo normal com nenhuma necessidade especial de escalonamento.
REALTIME PRIORITY CLASS	Indica um processo que tem a prioridade mais alta possível. Os encadeamentos de um processo de classe de prioridade de tempo real fazem a preempção de todos os outros processos, incluindo os processos do sistema operacional que estejam executando tarefas importantes. Por exemplo, um processo de tempo real que executa por mais tempo que um breve intervalo pode levar os caches de disco a não gravarem seus dados ou fazer com que o mouse não responda.

Além de criar um processo, *CreateProcess* também cria um objeto de encadeamento. A função *CreateProcess* cria o encadeamento com uma pilha inicial cujo tamanho ele descreve no cabeçalho de imagem do arquivo executável do programa especificado. O encadeamento inicia a execução no ponto de entrada da imagem. *Create-*

process cria o novo processo e os novos indicativos de encadeamento com direitos de acesso pleno. Para ambos os indicativos, se a função não fornece um descritor de segurança, seu programa pode usar o indicativo em qualquer função que requeira um indicativo de objeto desse tipo. Quando a função fornece um descritor de segurança, seu programa executa uma verificação de acesso em todos os usos subseqüentes do indicativo antes de lhe conceder o acesso. Se a verificação de acesso negar o acesso, o processo solicitante não poderá usar o indicativo para obter acesso ao encadeamento.

O Windows atribui um identificador de 32 bits ao processo. O identificador é válido até que o processo termine. Seu programa pode usá-lo para identificar o processo ou para especificar a função *OpenProcess* para abrir o indicativo para o processo. O Windows também atribui um identificador de encadeamento de 32 bits ao encadeamento inicial no processo. O identificador é válido até que o encadeamento termine e seu programa possa usá-.o para identificar de forma exclusiva o encadeamento dentro do sistema. Esses identificadores são retornados na estrutura *PROCESS_INFORMATION.*

Quando você especifica um nome de aplicativo nas strings *lpNomeAplic* ou *lpLinhaCmd*, não importa se o nome do aplicativo inclui ou não a extensão do arquivo, com uma exceção: um aplicativo baseado no MS-DOS ou no Windows cuja extensão de nome de arquivo seja .COM precisa incluir a extensão .COM. O encadeamento chamador pode usar a função *WaitForInputIdle* para aguardar até que o novo processo termine sua inicialização e esteja aguardando a entrada do usuário sem entrada pendente. Usar *WaitForInputIdle* pode ser útil para sincronização entre os processos-pai e filho, *pois CreateProcess* retorna sem aguardar que o novo processo termine sua inicialização. Por exemplo, o processo chamador deve usar *WaitForInputIdle* antes de tentar localizar uma janela associada com o novo processo

O método recomendado pela Microsoft para encerrar um processo é usar a função *ExitProcess*, pois ela notifica a proximidade de finalização a todas as bibliotecas de ligações dinâmicas (DLLs) associadas com o processo. Os outros modos de desligar um processo não notificam as bibliotecas de ligações dinâmicas associadas. Observe que, quando um encadeamento chama *ExitProcess*, a função termina os outros encadeamentos de processo sem lhes dar a oportunidade de executar qualquer código adicional (incluindo o código de terminação de encadeamento das bibliotecas de ligações dinâmicas associadas).

O Windows serializa *ExitProcess*, *ExitThread*, *CreateThread*, *CreateRemoteThread* e um processo que está iniciando (como resultado de uma chamada a *CreateProcess*) entre uma e outra dentro de um processo. Esses eventos somente podem acontecer um de cada vez em um espaço de endereço, o que significa que as seguintes restrições são aplicáveis:

- Durante a inicialização do processo e das rotinas de inicialização de biblioteca de ligações dinâmicas, seu programa pode criar novos encadeamentos, mas eles não iniciam a execução até que o Windows finalize a inicialização da biblioteca de ligações dinâmicas para o processo.
- Somente um encadeamento em um processo pode estar em uma rotina de inicialização ou de remoção da biblioteca de ligações dinâmicas de cada vez.
- A função *ExitProcess* não retorna até que algum encadeamento esteja em sua rotina de inicialização ou de remoção da biblioteca de ligações dinâmicas.

O processo criado permanece no sistema até que todos os encadeamentos dentro do processo terminem, e as chamadas à função *CloseHandle* tiverem fechado todos os indicativos para o processo e quaisquer de seus encadeamentos. As chamadas à *CloseHandle* precisam fechar os indicativos para o processo e o encadeamento principal. Se seu programa não precisar desses indicativos, será melhor fechá-los imediatamente após o Windows criar o processo.

Quando o último encadeamento em um processo terminar, os seguintes eventos ocorrerão:

- O Windows implicitamente fecha todos os objetos que o processo abriu.
- O status de finalização do processo (que *GetExitCodeProcess*) muda de seu valor inicial de *STILL_ACTIVE* para o status de finalização do último encadeamento que termina.
- O Windows define o objeto de encadeamento do encadeamento principal para o estado sinalizado, satisfazendo quaisquer encadeamentos que estivessem aguardando no objeto.
- O Windows define o processo de objeto para o estado sinalizado, satisfazendo quaisquer encadeamentos que estivessem aguardando pelo objeto.

Se o diretório atual na unidade C for *\CBIBLIA\BIBLIA*, haverá variável de ambiente chamada *=C*, cujo valor é *C:\CBIBLIA\BIBLIA*. Como observado na descrição anterior de *lpAmbiente*, essas informações de diretório atual para as unidades do sistema não propagarão automaticamente para um novo processo quando o pa-

râmetro *lpAmbiente* da função *CreateProcess* não for *NULL*. Um aplicativo precisa passar manualmente as informações do diretório atual para o novo processo. Para fazer isso, o aplicativo precisa criar explicitamente as strings da variável de ambiente =*X*, alfabetizá-las (pois o Windows NT e o Windows 95 usam um ambiente classificado) e, depois, colocá-las em um bloco de ambiente que a função *lpAmbiente* especifica. Tipicamente, as strings de variável de ambiente =*X* estarão na frente do bloco de ambiente, devido à classificação do bloco de ambiente mencionado anteriormente.

Um modo de obter a variável de diretório atual para uma unidade X é chamar a função *GetFullPathName* (*"X:"*,...), que permite que um aplicativo evite ter que examinar um bloco de ambiente. Se o caminho completo que *GetFullPathName* retorna for *X:*, seu programa não precisará passar esse valor adiante como dados do ambiente, pois o diretório-raiz será o diretório atual para a unidade *X* de um novo processo. A função *CreateProcess* retorna o indicativo que tem *PROCESS_ALL_ACCESS* para o objeto do processo.

O diretório atual que o parâmetro *lpDiretAtual* especifica é o diretório atual para o processo-filho. O diretório atual no parâmetro *lpLinhaCmd* é o diretório atual para o processo-pai.

Nota: Sob o Windows NT, quando uma função cria um processo com o sinalizador de prioridade CREATE_NEW_PROCESS_GROUP especificado, o Windows NT faz uma chamada implícita a SetConsoleCtrlHandler(NULL, True) em nome do novo processo; isso significa que o novo processo tem o sinal Ctrl+C desabilitado. Isso permite que as boas interfaces de comando tratem Ctrl+C elas mesmas e passem seletivamente esse sinal para um subprocesso. Ctrl+Break não é um sinal desabilitado, e o Windows NT pode usá-lo para interromper o processo e o grupo do processo.

1378 TERMINANDO OS PROCESSOS

Como você aprendeu, seus programas executarão dentro de um processo. Na dica anterior, você aprendeu como criar um processo. Exatamente como com a maioria das alocações ou criações que seus programas executam, é sua responsabilidade encerrar o processo quando ele completar suas funções. Você usará a função *ExitProcess* para fechar um processo que esteja em execução atualmente. A função *ExitProcess* finaliza um processo, e todos os seus encadeamentos e retorna para a localização chamadora, como mostrado aqui:

```
VOID ExitProcess(
   UINT  uCdgSaida     // código de saída para todos os encadeamentos
```

O parâmetro *uCdgSaida* especifica o código de saída para o processo e para todos os encadeamentos que o processo termina como resultado da chamada à *ExitProcess*. Você usa a função *GetExitCodeProcess* para recuperar o valor de saída do processo, e a função *GetExitCodeThread* para recuperar o valor de saída de um encadeamento.

Você sempre deve chamar *ExitProcess* para finalizar um processo. A função *ExitProcess* oferece um processo limpo de desligar, o que inclui chamar a função de ponto de entrada de todas as bibliotecas de ligações dinâmicas (DLLs) associadas com um valor que indica que o processo está se removendo da biblioteca de ligações dinâmicas. Se um processo chamar *TerminateProcess* para terminar, o processo não notificará as bibliotecas de ligações dinâmicas que o Windows pode ter associado com o processo.

Após todas as bibliotecas de ligações dinâmicas associadas terem executado qualquer valor de processo de terminação, a função *ExitProcess* termina o processo atual. Finalizar um processo tem os seguintes efeitos:

1. A função *ExitProcess* fecha todos os indicativos de objeto que o processo abriu.
2. Todos os encadeamentos no processo terminam sua execução.
3. O estado do objeto do processo se torna sinalizado, o que retoma a execução de quaisquer encadeamentos que estavam aguardando o processo terminar.
4. Os estados de todos os encadeamentos de processo se tornam sinalizados, o que retoma a execução de quaisquer encadeamentos que estavam aguardando a finalização de qualquer um dos encadeamentos que compõem o processo.
5. O status de finalização do processo muda de *STILL_ACTIVE* para o valor de saída do processo.

Finalizar um processo não faz o Windows terminar os processos-filho. Terminar um processo não necessariamente remove o objeto do processo do sistema operacional. O sistema exclui um objeto de processo quando o programa fecha o último encadeamento para o processo. O Windows serializa as funções *ExitProcess*, *ExitThread*, *CreateThread* e *CreateRemoveThread*. Adicionalmente, o Windows serializa um processo que está iniciando

(como resultado de uma chamada à *CreateProcess*) com os processos nomeados anteriormente dentro do processo chamador. Apenas um desses eventos pode acontecer em um espaço de endereço a um tempo, o que significa que o processo mantém as seguintes restrições:

- Durante a inicialização do processo e as rotinas de inicialização da biblioteca de ligações dinâmicas, o programa pode criar novos encadeamentos, mas eles não iniciam a execução até que a inicialização da biblioteca de ligações dinâmicas seja feita para o processo.
- Somente um encadeamento em um processo pode estar em uma inicialização de biblioteca de ligações dinâmicas ou rotina destacada de cada vez.
- *ExitProcess* não retorna até que algum dos encadeamentos esteja em sua rotina de inicialização ou de remoção da biblioteca de ligações dinâmicas.

GERANDO PROCESSOS-FILHO 1379

Seus programas podem usar o comando *CreateProcess* para começar a executar outro processo Quando você projetar aplicativos mais complexos, frequentemente encontrará situações em que deseja que outro bloco de código realize trabalho. Dentro de seus programas, você poderá chamar uma função para realizar tal trabalho. No entanto, as funções são seriais em natureza — isto é, seu código não pode continuar seu processamento até após uma função completar seu processamento.

Um método alternativo para fazer outro bloco de código realizar trabalho dentro de seus programas é criar um novo encadeamento dentro de seu processo e fazer o novo encadeamento ajudar com o processamento. Usar múltiplos encadeamentos permite que o código do seu programa continue o processamento enquanto o novo encadeamento executa o trabalho solicitado. Infelizmente, usar múltiplos encadeamentos geralmente causará problemas de sincronização quando seu encadeamento precisar ver os resultados do novo encadeamento. Você aprenderá mais sobre sincronização em dicas posteriores.

A outra abordagem é gerar um novo processo (chamado um *processo-filho*) para ajudar com o trabalho. Trabalhar com processos permite que seus programas continuem processando enquanto o processo-filho trabalha em um problema específico, ou permite que seus programas façam uma pausa enquanto um processo-filho trabalha no problema específico. Nas próximas dicas, você trabalhará com os processos-filho e com encadeamentos. Trabalhar com ambos lhe ajudará a compreender melhor as diferenças entre eles, e como você poderá usá-los dentro de seus programas.

À medida que seus programas Windows se tornarem mais complexos, você trabalhará mais e mais com componentes compartilhados e não-relacionados. Esses componentes podem ser, por exemplo, as bibliotecas de ligações dinâmicas, que executam dentro do processo atual do programa (*objetos no processo*), ou servidores de automação de incorporação e ligação de objetos (OLE), que executam fora do processo atual do programa (*objetos fora do processo*). Este livro não discutirá os objetos no processo ou fora do processo em detalhes.

TRABALHANDO MAIS COM OS PROCESSOS-FILHO 1380

Como você aprendeu, seus programas podem controlar o modo como interagem com os processos-filho. No entanto, quase todos os processos-filho requerem acesso para os dados contidos no espaço de endereço do processo-pai. Em geral, quando um processo-filho requer acesso aos dados no processo-pai, você deve rodar o processo-filho em seu próprio espaço de endereço e simplesmente dar ao processo-filho acesso a dados relevantes no espaço de endereço do processo-pai. Controlar o acesso ao espaço do endereço do processo-pai lhe permite proteger dados não-relevantes para o processo-filho de corrupção inadvertida. O Win32 lhe dá vários métodos para transferir dados entre diferentes processos: intercâmbio dinâmico de dados (DDE), incorporação e ligação de objetos (OLE), canalizações, mailslots e assim por diante. Um dos modos mais convenientes (e fáceis) de compartilhar dados é usar um *arquivo mapeado em memória.*

Um arquivo mapeado em memória é um arquivo especial que lhe permite reservar uma região de espaço de endereço e comprometer espaço de memória física para a região, similar à alocação de memória virtual. Ao contrário da memória virtual, no entanto, o armazenamento físico do arquivo mapeado na memória vem a partir de um arquivo que já está no disco, e não do arquivo de paginação do sistema. Após você mapear o arquivo, poderá acessar o arquivo inteiro como se seu programa tivesse carregado na memória.

Você usará os arquivos mapeados na memória para os três propósitos a seguir:

- O sistema usa arquivos mapeados em memória para carregar e executar arquivos executáveis e da biblioteca de ligações dinâmicas. Usar arquivos mapeados em memória conserva grandemente tanto o espaço no arquivo de paginação quanto o tempo que um aplicativo requer para começar a executar.
- Você pode usar arquivos mapeadas em memória para acessar um arquivo de dados no disco. Usar um arquivo mapeado em memória lhe protege de executar operações de E/S em um arquivo e bufferizar o conteúdo do arquivo.
- Você pode usar arquivos mapeados em memória para permitir que múltiplos processos que estejam rodando na mesma máquina compartilhem dados um com o outro, como você já aprendeu.

Muitos objetos de comunicações Win32 usarão a estrutura de arquivo mapeado em memória lhe porque é poderoso e fácil de usar. Você aprenderá mais sobre como criar um arquivo mapeado em memória em dicas posteriores.

Se você quiser criar um novo processo e fazer com que ele execute algum trabalho enquanto o programa-pai aguarda pelo resultado (por exemplo, escrever algum dado em um arquivo que o processo-pai mais tarde lerá), o processo-pai pode usar código similar ao seguinte:

```
PROCESS_INFORMATION pi;
DWORD dwCdgSaida;

BOOL fSucesso = CreateProcess(NomeProcesso, &pi);
if (fSucesso)
 {
   // Fecha o indicativo do encadeamento assim que você
   // não precisar mais dele
   CloseHandle(pi.hEncadeam);
   WaitForSingleObject(pi.hProcesso, INFINITO);
   // Processo terminado

   GetExitCodeProcess(pi.hProcesso, &dwCdgSaida);
   // Fecha o indicativo do processo.
   CloseHandle(pi.hProcesso);
 }
```

O fragmento de código cria o novo processo *NomeProcesso*. Se for bem-sucedido, o fragmento fechará o indicativo do encadeamento extra para liberar tempo da CPU, depois espera que o processo complete seu processamento. Após o processo terminar, a variável *dwCdgSaida* conterá as informações de código de saída do programa, e o fragmento de código fechará o indicativo do processo. Se não for bem-sucedido ao iniciar o processo, o fragmento de código não executará nenhum processamento.

1381 RODANDO PROCESSOS-FILHO DESTACADOS

Na dica anterior, você aprendeu como criar um processo-filho e parar a execução do encadeamento atual até que o processo terminasse. No entanto, na maioria das vezes, seus programas iniciarão outro proceso como um *processo-filho destacado*. Um processo-filho destacado é um processo que a execução do processo-pai, após o processo-pai criar o processo-filho e o processo-filho iniciar a execução, não requer mais comunicação com o processo-filho ou não requer que o processo-filho finalize seu trabalho antes que o processo-pai continue. Rodar processos-filho destacados permite que seus programas rodem outros programas sem se preocupar com seus tempos de administração ou com o desempenho. Por exemplo, quando você executa um programa a partir de dentro do *Explorer*, o *Explorer* cria um novo processo; depois ignora o novo processo e continua seu próprio processamento.

Quando seus programas criam um processo-filho destacado, o programa precisa primeiro criar o processo, depois fechar seus indicativos para o novo processo e seu encadeamento primário. O fragmento de código a seguir mostra como seus programas podem criar um novo processo-filho destacado:

```
BOOL fSucesso = CreateProcess(NomeProcesso, &pi);
if (fSucesso)
  {
     CloseHandle(pi.h.Encadeam);
     CloseHandle(pi.hProcesso);
  }
```

COMPREENDENDO MELHOR OS ENCADEAMENTOS 1382

Em dicas anteriores você aprendeu como criar e gerenciar os processos. Cada processo do Win32 contém um ou mais encadeamentos. Lembre-se, um encadeamento é um caminho de execução dentro de um processo. Toda vez que o Windows inicializa uma nova ocorrência de um processo, o sistema operacional cria um novo encadeamento primário para esse processo. O encadeamento primário inicia quando o Windows carrega o programa. O encadeamento, por sua vez, chamará sua função *WinMain* e continuará em execução até que sua função *WinMain* termine seu processamento e o programa chame *ExitProcess* para finalizar a si mesmo. Para muitos aplicativos, o encadeamento primário que o sistema operacional cria é o único encadeamento que o aplicativo requer No entanto, os processos podem criar encadeamentos adicionais para ajudá-los a fazer seu trabalho. A idéia por trás da criação de encadeamentos adicionais é usar o máximo de tempo de processamento da CPU tanto quanto possível e tão eficientemente quanto possível. Quando você criar encadeamentos adicionais, enviará solicitações adicionais de tempo de CPU para o sistema operacional. Como você aprenderá em dicas posteriores, os encadeamentos adicionais permitirão que seus programas efetuem processamento em segundo plano de forma mais eficiente, façam melhores cálculos e efetuem atividades com base em tempo e em eventos, bem como muitas outras tarefas avançadas de programação. As Dicas 1383 e 1384, a seguir, discutirão em detalhes quando criar e quando não criar um encadeamento.

Quando você pensa sobre o que está atualmente executando no sistema, é útil visualizar os encadeamentos como encapsulados dentro dos processos. Dependendo da prioridade do encadeamento, o sistema processará alguns encadeamentos com mais frequência do que outros, tanto entre diferentes processos quanto dentro de um único processo. A Figura 1382 mostra um modelo lógico de vários processos, cada um dos quais contém múltiplos encadeamentos rodando no sistema operacional Windows, bem como uma ordem potencial em que a CPU poderá processar esses encadeamentos.

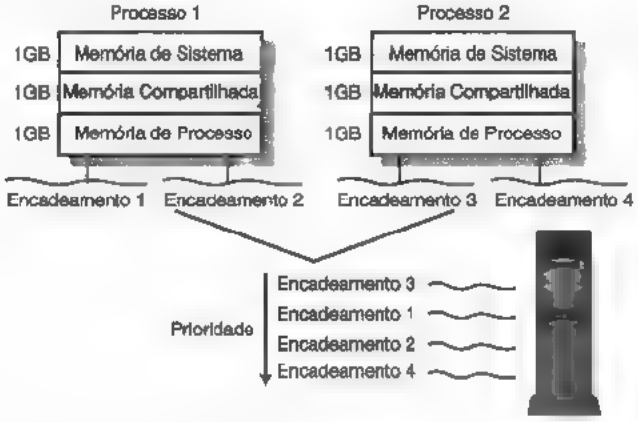

Figura 1382 O modelo lógico processo-encadeamento.

AVALIANDO A NECESSIDADE DE ENCADEAMENTOS 1383

Vimos na dica anterior que seus programas usarão com mais freqüência encadeamentos para garantir que várias seções de um determinado aplicativo recebam o máximo de acesso possível à CPU do computador. Determinar quando ou não usar encadeamentos adicionais é uma das decisões mais importantes que você fará quando codificar programas Windows.

Por exemplo, considere um programa de planilha. Ele precisa recalcular sempre que o usuário fizer entrada de dados dentro das células. Como o recálculo para planilhas complexas pode demorar vários segundos para completar, um aplicativo bem projetado não deverá recalcular a planilha após cada modificação que o usuário fizer. Em vez disso, o aplicativo deverá executar a função de recálculo da planilha como um encadeamento separado com uma prioridade menor que a do encadeamento primário. Se você usar dois encadeamentos, o primário sempre rodará enquanto o usuário estiver digitando, de modo que o encadeamento de recálculo de baixa prioridade não acessará a CPU. Quando o usuário parar de digitar, o encadeamento de prioridade mais baixa executará enquanto o encadeamento primário aguarda que o usuário comece a digitar de novo.

Claramente, você pode aplicar melhor a utilidade dos encadeamentos para programas complexos que efetuam múltiplos conjuntos de tarefas. As atividades de segundo plano também fazem excelente uso de encadeamentos adicionais. Outras situações que fazem bom uso dos encadeamentos incluem:

- É útil criar um encadeamento separado para tratar as tarefas de impressão dentro de um aplicativo, o que permita que o usuário continue a usar o aplicativo enquanto este executa a tarefa de impressão.
- Você pode usar um encadeamento separado para manter uma caixa de diálogo não-modal que permita que o usuário interrompa tarefas estendidas, tais como as de cópia ou de impressão.
- Você pode usar os encadeamentos para criar aplicativos que simulem eventos do mundo real (por exemplo, eventos que ocorram em um intervalo de tempo predeterminado).

1384 DETERMINANDO QUANDO NÃO CRIAR UM ENCADEAMENTO

A primeira vez que muitos programadores obtêm acesso a um ambiente que suporta múltiplos encadeamentos, *eles tendem a abusar do uso de encadeamentos simplesmente devido ao novo poder de processamento e a usabilidade* que os encadeamentos permitem que eles explorem. Muitos programadores começam dividindo os aplicativos existentes em partes menores, cada uma das quais executa seu próprio encadeamento. A despeito de quão útil esse processamento com base em encadeamento possa parecer a princípio, ele pode, na verdade, resultar em muito tempo perdido de processamento. Cada encadeamento requer uma certa quantidade de sobrecarga para criar, assim como cada encadeamento na fila, mesmo se o sistema operacional não liberar o encadeamento para a CPU para processamento. Os encadeamentos adicionais reduzirão o desempenho do sistema porque, além da sobrecarga do encadeamento, o sistema precisa conferir o nível de prioridade de cada encadeamento atualmente ativo ao determinar qual encadeamento executar.

Como você aprendeu, os encadeamentos são incrivelmente úteis e importantes e têm um lugar em todos os programas Windows. No entanto, é importante reconhecer que quando usa encadeamentos, você pode potencialmente criar novos problemas ao tentar resolver os antigos. Por exemplo, se estiver desenvolvendo um programa de processamento de texto e quiser fazer a função de impressão rodar como seu próprio encadeamento, sua resposta inicial poderá ser criar esse encadeamento e imprimir a página de documento página por página. Infelizmente, o usuário poderia alterar o documento enquanto você o está imprimindo no segundo plano. Em vez da solução simples que você poderia esperar, você precisará em seu lugar, copiar o arquivo que está sendo impresso para um arquivo temporário, imprimir e depois excluir o arquivo temporário. Usar múltiplos encadeamentos e um arquivo temporário é, sem dúvida, mais eficiente para o usuário e torna o programa mais atraente para o usuário. No entanto, você precisa ter cuidado para garantir que criar encadeamentos adicionais não põe em perigo o processamento que seu programa executa dento de seus encadeamentos atuais.

Existem várias regras que você deve aplicar ao determinar se deve ou não criar múltiplos encadeamentos:

- Todos os componentes da interface do usuário (controles e janelas) devem compartilhar um encadeamento comum, com raríssimas exceções.
- Os programas somente devem criar encadeamentos quando precisarem deles, e não criar encadeamentos adicionais e "mantê-los na reserva".
- Os programas devem liberar os encadeamentos assim que o processamento do encadeamento terminar.
- Os aplicativos mais complexos, que usam múltiplas janelas, podem requerer encadeamentos extras para tratar o processamento dentro de certas janelas.
- Você não deve criar encadeamento que pode permitir que o usuário interrompa um processo crítico do sistema e, portanto, corromper a memória ou outros processos atualmente em execução.

Como regra, você sempre deve ter certeza de que precisa de um encadeamento ou de que o encadeamento melhorará significativamente o processamento do programa. Usar encadeamentos adicionais sem fazer essa determinação levará a desperdício de tempo de processador e à redução na velocidade de execução dos seus programas.

1385 CRIANDO UMA FUNÇÃO SIMPLES DE ENCADEAMENTO

Em dicas anteriores, você aprendeu que seus programas podem usar múltiplos encadeamentos para permitir que seus programas efetuem múltiplas atividades de execução dentro de um único processo. Quando você criar encadeamentos, seus programas geralmente usarão a função *CreateThread*. A função *CreateThread* cria um encadea-

mento para executar dentro do espaço de endereço do processo chamador. Você usará a função *CreateThread* dentro de seus programas, como mostra o protótipo a seguir:

```
HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpAtributoEncad, // ponteiro para atributos
                                          de segurança do encadeamento
   DWORD dwTamPilha, // tamanho inicial da pilha do encadeamento
   LPTHREAD_START_ROUTINE lpEnderInicial, // ponteiro para função de
                                          encadeamento
   DWORD dwSinalizCriacao,     // sinalizadores de criação
   LPDWORD lpIdentEncad // ponteiro para o identificador de encadeamento
                        retornado
);
```

A função *CreateThread* aceita os parâmetros detalhados na Tabela 1385.

Tabela 1385 Os parâmetros para a função *CreateThread*.

Parâmetro	Descrição
lpAtributoEncad	Ponteiro para uma estrutura *SECURITYATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtributoEncad* for *NULL*, os processos-filho não poderão herdar o indicativo. Sob o Windows NT, o membro *lpDescritorSeg* da estrutura especifica um descritor de segurança para o novo encadeamento. Se *lpAtributoEncad* for *NULL*, o encadeamento obterá um descritor de segurança padrão. Sob o Windows 95, a função *CreateThread* ignora o membro *lpDescritorSeg* da estrutura.
dwTamPilha	Especifica o tamanho, em bytes, da pilha para o novo encadeamento Se *dwTamPilha* especificar zero, o tamanho da pilha assumirá como padrão o mesmo tamanho que o do encadeamento primário do processo. O Windows automaticamente aloca a pilha do processo no espaço de memória do processo e libera a pilha quando o encadeamento termina. Observe que o tamanho da pilha cresce, se necessário. *CreateThread* tenta comprometer o número de bytes que *dwTamPilha* especifica, e falhará se o tamanho exceder a memória disponível.
lpEnderInicial	O endereço inicial do novo encadeamento. Isso é tipicamente o endereço de uma função declarada com a convenção de chamada WINAPI que aceita um único ponteiro de 32 bits como um argumento e retorna um código de saída de 32 bits. Seu protótipo é mostrado aqui: *DWORD WINAPI FuncaoEncad(LPVOID);*
lpParametro	Especifica um único valor de parâmetro de 32 bits passado para o encadeamento.
dwSinalCriacao	Especifica sinalizadores adicionais que controlam a criação do encadeamento. Se *dwSinalizCriacao* especificar o sinalizador *CREATE SUSPENDED*, o processo criará o encadeamento em um estado suspenso e não rodará o encadeamento até que o programa (ou outro programa) chame a função *ResumeThread*. Se o parâmetro for zero, o encadeamento rodará imediatamente após a criação. O Windows atualmente não suporta outros valores.
lpEncadeam	Aponta para uma variável de 32 bits que recebe o identificador de encadeamento.

Se a função *CreateThread* for bem-sucedida, o valor de retorno será um indicativo para o novo encadeamento Se a função falhar, o valor de retorno será *NULL*. Sob o Windows 95, *CreateThread* é bem-sucedida somente quando o sistema a chama no contexto de um programa de 32 bits. Uma biblioteca de ligações dinâmicas de 32 bits não pode criar um encadeamento adicional quando um programa de 16 bits a chama.

A função *CreateThread* cria o novo indicativo de encadeamento com acesso total para o novo encadeamento. Se a chamada à *CreateThread* não fornece um descritor de segurança, o programa pode usar o indicativo retornado em qualquer função que requer um indicativo de objeto de encadeamento. Quando o processo não fornece um descritor de segurança, o programa efetua uma verificação de acesso em todos os usos subseqüentes

do indicativo antes de permitir o acesso. Se a verificação de acesso nega o acesso, o processo que faz a solicitação *não pode usar o indicativo para obter acesso ao encadeamento.*

A execução do encadeamento inicia na função que o parâmetro *lpEnderInicial* especifica. Se essa função retornar, *CreateThread* usará o valor de retorno *DWORD* para terminar o encadeamento em uma chamada implícita à função *ExitThread*. Seus programas devem usar a função *GetExitCodeThread* para obter o valor de retorno do encadeamento após a finalização do encadeamento.

A função *CreateThread* poderá ser bem-sucedida mesmo se *lpEnderInicial* apontar para código ou para dados ou se ele não estiver acessível. Se o endereço inicial for inválido quando o encadeamento rodar, uma exceção ocorrerá e o encadeamento terminará. O programa trata a finalização do encadeamento devido a um endereço inicial inválido como um erro de saída para o processo do encadeamento. Esse comportamento é similar à natureza assíncrona de *CreateProcess*, onde a função cria o processo mesmo se ele se referir à biblioteca de ligações dinâm.cas (DLLs) inválidas ou ausentes.

A função *CreateThread* cria o encadeamento com uma prioridade de *THREAD_PRIORITY_NORMAL*. Use as funções *GetThreadPriority* e *SetThreadPriority* para obter e definir o valor de prioridade de um encadeamento.

O objeto de encadeamento permanece no sistema até que o encadeamento tenha terminado e o programa tenha fechado todo os encadeamentos para ele com uma chamada à *CloseHandle*. O programa serializa as funções *ExitProcess*, *ExitThread*, *CreateThread* e *CreateRemoteThread* e um processo que está iniciando (como resultado de uma chamada à *CreateProcess*) entre um e o outro dentro de um processo. Somente um desses eventos pode ocorrer em um espaço de endereço de cada vez, o que significa que o processo mantém as seguintes restrições:

- Durante as rotinas de inicialização do processo e das bibliotecas de ligações dinâmicas, o programa pode criar novos encadeamentos, mas eles não iniciam a execução até que o processo complete a inicialização da biblioteca de ligações dinâmicas.
- Somente um encadeamento em um processo pode estar em uma rotina de inicialização ou de remoção de uma biblioteca de ligações dinâmicas de cada vez.
- *ExitProcess* não retorna até que algum encadeamento esteja em suas rotinas de inicialização ou de remoção de uma biblioteca de ligações dinâm.cas.

Nota: Um encadeamento que usa funções das bibliotecas de tempo de execução de C devem usar as funções de execução ***beginthread*** *e* ***endthread*** *para o gerenciamento de encadeamento em vez de* ***CreateThread*** *e* ***ExitThread****. Deixar de fazer isso resulta em pequenas perdas de memória quando o programa chama* ***ExitThread****.*

Para compreender melhor o processamento que a função *CreateThread* executa, considere o programa *Simple_Thread.cpp* contido no CD-ROM que acompanha este livro. Esse programa cria um novo encadeamento toda vez que o usuário selecionar a opção de menu *Testar!* e exibe na janela que o encadeamento in.ciou. Quando você selecionar a opção *Sair*, o programa liberará cada encadeamento um de cada vez e lhe alertará da destruição do encadeamento. O código dentro da função *WndProc* cria o encadeamento, como mostrado aqui:

```
case IDM_TEST :          // inicia um encadeamento.
   {
     DWORD dwIdFilha;
     CreateThread(NULL, 0, ChildThreadProc, hJan, 0, &dwIdFilha);
   }
   break;
```

1386 VISUALIZANDO A INICIAÇÃO DO ENCADEAMENTO

Seus programas podem criar encadeamentos para efetuar processamento adicional dentro de seu próprio espaço de execução. Toda vez que você cria um novo encadeamento, o Windows executa várias tarefas básicas para inicializar o encadeamento e iniciar sua execução.

Primeiro, o Windows aloca em cada encadeamento sua própria pilha. O Windows aloca a pilha no encadeamento a partir do espaço de endereço de 4Gb do processo proprietário. Quando seus programas usam variáveis estáticas e globais, múltiplos encadeamentos podem acessar essas variáveis ao mesmo tempo, potencialmente corrompendo o conteúdo da variável. No entanto, o Windows cria variáveis locais e automáticas na pilha do encadeamento, portanto, tornando-as menos prováveis de sofrer corrupção em programas multien-

cadeados do que as variáveis globais. Como você aprendeu, sempre deve tentar usar variáveis locais e automáticas, em vez de variáveis globais. Essa regra se aplica até com mais força quando você trabalha com encadeamentos.

Segundo, o Windows aloca em cada encadeamento seu próprio conjunto de registradores da CPU, chamados *contexto* do encadeamento. O Windows armazena o contexto do encadeamento dentro da estrutura *CONTEXT*. Seus programas podem consultar a estrutura *CONTEXT* em qualquer tempo para determinar o estado dos registradores da CPU do encadeamento. Quando o sistema operacional escalona o tempo da CPU para um encadeamento, o sistema inicializa os registradores da CPU com o contexto do encadeamento. Os registradores da CPU incluem um ponteiro da instrução (que identifica o endereço da próxima instrução da CPU para o encadeamento executar) e um ponteiro de pilha (que identifica o endereço da pilha do encadeamento).

Após o encadeamento completar essas inicializaçoes da pilha e do contexto, o encadeamento (assumindo que você não o criou em um estado suspenso) iniciará sua execução na primeira linha da função que você definiu dentro da criação do encadeamento.

PASSOS QUE O SISTEMA OPERACIONAL EFETUA NA CRIAÇÃO DO ENCADEAMENTO 1387

Como você aprendeu na dica anterior, o sistema operacional efetua diversas tarefas importantes quando aloca encadeamentos, as quais seus programas utilizarão. No entanto, o sistema operacional na verdade realiza seis passos específicos sempre que cria um novo encadeamento. São eles:

1. Aloca um objeto de núcleo do encadeamento para identificar e gerenciar o encadeamento recém-criado. O objeto de núcleo contém muitas das informações do sistema para gerenciar o encadeamento. Um indicativo para o objeto de núcleo do encadeamento é o valor que *CreateThread* retorna.
2. Inicializa o código de saída do encadeamento (que o Windows mantém no objeto de núcleo do encadeamento) como *STILL_ACTIVE* e define o contador de suspensão do encadeamento (que o Windows também mantém no objeto de núcleo do encadeamento) como 1.
3. Aloca uma estrutura *CONTEXT* para o novo encadeamento.
4. Prepara a pilha do encadeamento reservando uma região de espaço de endereço, comprometendo duas páginas de armazenagem física para a região, definindo a proteção da armazenagem comprometida como *PAGE_READWRITE* e definindo o atributo *PAGE_GUARD* na segunda página a partir do topo.
5. Coloca os valores *lpEnderInicial* e *lpvEncadeam* no topo da pilha para que o novo encadeamento os veja como parâmetros passados para a função *StartOfThread* (somente se seu código usa a biblioteca de execução de C).
6. Inicializa o registrador Ponteiro da Pilha (SP) na estrutura *CONTEXT* no encadeamento para apontar para os valores que o Windows colocou na pilha no Passo 5. Em seguida, o sistema operacional inicializa o registrador SP para apontar para a função interna que o Windows processa antes de executar a primeira instrução na função de inicialização do encadeamento.

DETERMINANDO O TAMANHO DA PILHA DO ENCADEAMENTO 1388

Na Dica 1385 foi visto que seus programas podem especificar o tamanho da pilha do encadeamento dentro da função *CreateThread*. É importante notar que após o Windows criar o encadeamento, seu programa não pode modificar com segurança o tamanho da pilha do encadeamento. Em vez disso, o Windows aumentará dinamicamente o tamanho da pilha para baixo, conforme necessário.

Se você não especificar um tamanho para a pilha do encadeamento, o Windows alocará uma pilha do mesmo tamanho que o tamanho do encadeamento principal. *CreateThread* criará a pilha dentro do espaço de endereço de memória do processo. Você pode visualizar melhor a criação da pilha em um modelo lógico do espaço de pilha alocada, como mostra a Figura 1388.

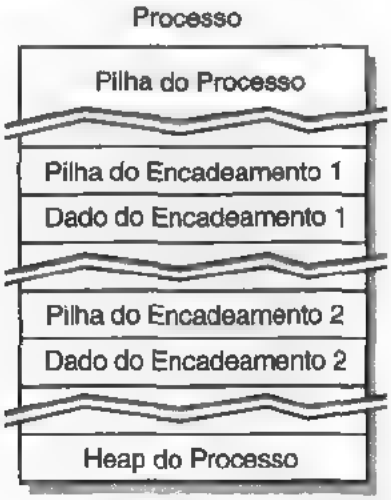

Figura 1388 Espaço da pilha para o processo, o encadeamento primário e um secundário.

Quando o Windows aloca espaço na pilha para encadeamentos adicionais, ele geralmente faz isso abaixo do espaço de pilha do processo, e a um segmento de distância. O Windows alocará o espaço de pilha do encadeamento virtualmente para que possa mover a pilha do encadeamento conforme for necessário. Por exemplo, se o processo inicializa o encadeamento secundário, depois cria uma matriz automática multidimensional muito grande (digamos, [1024,64] de caracteres), o Windows precisa poder mover a pilha secundária para evitar que o processo primário sobrescreva acidentalmente a pilha secundária.

1389 Obtendo um Indicativo para o Processo ou o Encadeamento Atual

À medida que seus programas se tornarem mais complexos, algumas vezes eles precisarão obter dinamicamente um indicativo para o encadeamento ou processo atual. Efetuar uma das atividades é relativamente simples. Seus programas podem chamar a função *GetCurrentThread* ou *GetCurrentProcess* para obter o pseudo-indicativo do encadeamento ou do processo atual (ele é chamado *pseudo-indicativo* porque seu valor somente é significativo dentro do encadeamento ou do processo atual) a qualquer tempo dentro de seus programas. Você implementará essas funções de acordo com seus protótipos, como mostrado aqui:

```
HANDLE GetCurrentThread(VOID);
HANDLE GetCurrentProcess(VOID);
```

Você deve notar que os *pseudos-indicativos* que ambas as funções retornam são inúteis fora do processo atual. Para passar o indicativo de um processo ou de um encadeamento para outro processo, seus programas precisam usar a função *DuplicateHandle*.

Para compreender melhor o processamento que as funções *GetCurrentThread* e *GetCurrentProcess* executam, considere o programa *Show_Current.cpp*, contido no CD-ROM que acompanha este livro. O programa *Show_Current.cpp* cria encadeamentos um de cada vez, e exibe informações sobre os encadeamentos e o processo.

1390 Tratando o Tempo de Processamento do Encadeamento

Como você pode imaginar, determinar o tempo que seu processo levará para executar uma determinada atividade em um ambiente multiencadeado será significativamente mais difícil do que no ambiente de um único encadeamento. Como seu processo poderá ter um encadeamento que esteja ocupado recalculando um algoritmo complexo enquanto os encadeamentos em outros processos continuarem a competir por tempo na CPU, seu processo poderá levar consideravelmente mais para executar entre duas situações de teste de avaliação. Seus pro-

gramas precisarão usar um método diferente para registrar tempo de processamento do encadeamento do código simples para tempo de execução do programa, como você fez na Dica 625 com o programa *clock.c*.

Em vez disso, seus programas precisam usar uma função que verifica o tempo da execução de um encadeamento. Dentro do Windows, você usará a função *GetThreadTimes* para efetuar esse processamento. A função *GetThreadTimes* obtém informações de tempo sobre um encadeamento que você especificar. Você usará a função *GetThreadTimes* dentro de seus programas, como mostra o seguinte protótipo:

```
BOOL GetThreadTimes(
   HANDLE hEncad,              // especifica o encadeamento de interesse
   LPFILETIME lpHoraCriacao, // quando o encadeamento foi criado
   LPFILETIME lpHoraSaida,   // quando o encadeamento foi destruído
   LPFILETIME lpTempoNucleo, // tempo que o encadeamento gastou no modo do
                                 núcleo
   LPFILETIME lpTempoUsuario // tempo que o encadeamento gastou no modo
                                 usuário
);
```

A função *GetThreadTimes* aceita os parâmetros detalhados na Tabela 1390.

Tabela 1390 Os parâmetros para a função ***GetThreadTimes***

Parâmetro	Descrição
hEncadem	Um indicativo de abertura que especifica o encadeamento cujas informações de horário um programa procura. Você precisa criar esse indicativo com o acesso *THREAD QUERY_INFORMATION*
lpHoraCriacao	Aponta para uma estrutura *FILETIME* que recebe a hora de criação do encadeamento.
lpHoraSaida	Aponta para uma estrutura *FILETIME* que recebe a hora de saída do encadeamento Se o encadeamento não terminou, o conteúdo dessa estrutura é indefinido.
lpTempoNucleo	Aponta para uma estrutura *FILETIME* que recebe a quantidade de tempo que o encadeamento executou no modo núcleo.
lpTempoUsuario	Aponta para uma estrutura *FILETIME* que recebe a quantidade de tempo que o encadeamento executou no modo usuário.

Se a função for bem-sucedida, o valor de retorno será diferente de zero. Caso contrário, o valor será zero. A função *GetThreadTimes* usa estrutura de dados *FILETIME* para expressar todos os tempos. Essas estruturas contêm dois valores de 32 bits que combinam para formar um contador de 64 bits de unidades de tempo de 100 nanossegundos. Os horários da criação e de finalização de encadeamentos são pontos em tempo que a estrutura *FILETIME* expressa como a quantidade de tempo que passou desde a meia-noite de 1 de janeiro de 1601 no horário de Greenwich. A API Win32 fornece várias funções que um aplicativo pode usar para converter esses valores em forma geralmente mais úteis.

Os tempos dos encadeamentos no modo núcleo e no modo usuário são quantidades de tempo medidas em nanossegundos. Por exemplo, se um encadeamento gastar um segundo no modo núcleo, a função *GetThreadTimes* preencherá a estrutura *FILETIME* que o parâmetro *lpTempoNucleo* especifica com um valor de 64 bits de dez milhões, que é o número de unidades de 100 nanossegundos em um segundo.

GERENCIANDO O TEMPO DE PROCESSAMENTO DE MÚLTIPLOS ENCADEAMENTOS 1391

Como você aprendeu na dica anterior, seus programas podem usar a função *GetThreadTimes* para determinar o tempo de execução de um encadeamento. No entanto, com frequência seus programas irão requerer informações detalhadas sobre o tempo de execução de muitos encadeamentos dentro de um processo (para determinar se todos os encadeamento dentro dele são lentos, ou se um ou mais encadeamento é a causa de uma lentidão do processo).

Nesses casos, seus programas podem usar a função *GetProcessTimes* para obter informações de tempo sobre um processo que você especificar. Você usará a função *GetProcessTimes* dentro de seus programas, como segue:

```
BOOL GetProcessTimes(
  HANDLE hProcesso,          // especifica o processo de interesse
  LPFILETIME lpHoraCriacao, // quando o processo foi criado
  LPFILETIME lpHoraSaida,   // quando o processo terminou
  LPFILETIME lpHoraNucleo,  // tempo que o processo gastou no modo núcleo
  LPFILETIME lpTempoUsuario // tempo que o processo gastou no modo usuário
```

A função *GetProcessTimes* aceita os parâmetros detalhados anteriormente na Tabela 1390.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, a função retornará um valor zero. A função *GetProcessTimes* usa estruturas da dados *FILETIME* para expressar todos os tempos. Essas estruturas contêm dois valores de 32 bits que se combinam para formar um contador de 64 bits de unidades de tempo de 100 nanossegundos. Os tempos de criação e saída do processo são pontos em tempo que a estrutura *FILETIME* expressa como a quantidade de tempo que transcorreu desde a meia-noite de 1 de janeiro de 1601 no horário de Greenwich. A API Win32 fornece várias funções que um aplicativo pode usar para converter esses valores em formas mais úteis.

Os tempos do modo núcleo e modo usuário do processo são quantidades de tempo. Por exemplo, se um processo gastou um segundo no modo núcleo, a função *GetProcessTimes* preencherá a estrutura que o parâmetro *lpHoraNucleo* especifica com um valor de 64 bits de dez milhões, o número de unidades de 100 nanossegundos em um segundo.

O CD-ROM que acompanha este livro contém o programa *Show_ThPr_Times*, que abre uma série de encadeamentos e retorna o tempo de processamento de cada encadeamento, bem como o tempo de processamento do programa inteiro.

1392 COMPREENDENDO MELHOR A FUNÇÃO GETQUEUESTATUS

À medida que você trabalha com encadeamentos, encontrará situações em que eventos ocorrem (tais como digitações e assim por diante) que o encadeamento primário normalmente processará dentro da função de mensagem *ProcJan* do seu programa. No entanto, se você tiver um encadeamento-filho suspenso, o controle sobre as informações de mensagens se tornará mais complicado. No entanto, como o encadeamento já está suspenso (por uma razão qualquer), a função de mensagem do encadeamento reterá essas mensagens para aquele encadeamento na fila de mensagem do encadeamento. Para determinar o conteúdo da fila de mensagem após um encadeamento suspenso reiniciar sua execução, seus programas podem usar a função *GetQueue* Status. A função *GetQueueStatus* retorna sinalizadores que indicam os tipos de mensagens encontrados na fila de mensagem do encadeamento chamador. Você implementará a função *GetQueueStatus* dentro de seus programas de acordo com o seguinte protótipo:

```
DWORD GetQueueStatus(UINT sinaliz);
```

O parâmetro *sinaliz* especifica sinalizadores de status da fila que dão os tipos de mensagens os quais a função deve checar. Este parâmetro pode ser uma combinação dos valores listados na Tabela 1392.

Tabela 1392 Valores de sinalizadores possíveis para o parâmetro ***sinaliz***.

Valor	Significado
QS_ALLEVENTS	Uma entrada, *WM_TIMER*, *WM_PAINT*, *WM_HOTKEY*, ou mensagem encaminhada está na fila.
QS_ALLINPUT	Qualquer mensagem está na fila.
QS_HOTKEY	Uma mensagem *WM_HOTKEY* está na fila.
QS_INPUT	Uma mensagem de entrada está na fila.
QS_KEY	Uma mensagem *WM_KEYUP*, *WM_KEYDOWN*, *WM_SYSKEYUP* ou *WM_SYSKEYDOWN* está na fila.

Tabela 1392 Valores de sinalizadores possíveis para o parâmetro *sinaliz*. (*Continuação*)

Valor	Significado
QS_MOUSE	Uma mensagem *WM_MOUSEMOVE* ou mensagem de botão do mouse (*WM_LBUTTONUP*, *WM_RBUTTONDOWN* e assim por diante) está na fila.
QS_MOUSEBUTTON	Uma mensagem de botão do mouse (*WM_LBUTTONUP*, *WM_RBUTTONDOWN* e assim por diante) está na fila.
QS_MOUSEMOVE	Uma mensagem *WM_MOUSEMOVE* está na fila.
QS_PAINT	Uma mensagem *WM_PAINT* está na fila.
QS_POSTMESSAGE	Uma mensagem encaminhada (diferente das que acabam de ser listadas) está na fila.
QS_SENDMESSAGE	Uma mensagem que outro encadeamento ou aplicativo enviou está na fila.
QS_TIMER	Uma mensagem *WM_TIMER* está na fila.

A palavra mais significativa do valor de retorno indica os tipos de mensagens atualmente na fila. A palavra menos significativa indica os tipos de mensagens que o Windows acrescentou na fila e que ainda estão na fila desde a última chamada à *GetQueueStatus*, *GetMessage* ou *PeekMessage*.

A presença de um sinalizador *QS_* no valor de retorno não garante que uma chamada subseqüente à função *PeekMessage* ou *GetMessage* retornará uma mensagem. *GetMessage* e *PeekMessage* efetuam alguma filtragem interna que pode fazer o programa processar a mensagem internamente. Por essa razão, você deve considerar o valor de retorno de *GetQueueStatus* somente como uma dica se seu programa chamar *GetMessage* ou *PeekMessage*.

Para compreender melhor o processamento que a função *GetQueueStatus* executa, considere o programa *Queue_Status.cpp*, contido no CD-ROM que acompanha este livro. O programa *Queue_Status* cria um encadeamento, depois adormece por cinco segundos para que o usuário possa gerar eventos que o Windows encaminha para a fila de mensagens. Após o encadeamento reiniciar, o programa exibirá os eventos dentro da fila.

TRATANDO AS EXCEÇÕES NÃO-TRATADAS 1393

O sistema operacional Win32 coloca uma rotina de tratamento de exceção no início de cada encadeamento e processo. O propósito da rotina de tratamento de exceção é garantir que os programas respondam educadamente às exceções não-tratadas (essencialmente, que os programas desliguem sem causar impacto negativamente em outros processos). No entanto, algumas vezes, você pode querer capturar todas as exceções não-tratadas dentro de uma rotina especial, talvez uma que poupe o trabalho do usuário de trabalhar em um arquivo de recuperação no disco. A função *SetUnhandledExceptionFilter* permite que um aplicativo substitua a rotina de tratamento de exceção de alto nível que o Win32 coloca no início de cada encadeamento e processo.

Após chamar a função *SetUnhandledExceptionFilter*, se uma exceção ocorrer em um processo que o Windows não esteja depurando no momento, e a exceção chegar ao filtro de exceções não-tratadas de Win32, esse filtro chamará a função de filtro de exceção especificada pelo parâmetro *lpAltoNivelFiltroExcecao*:

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
   LPTOP_LEVEL_EXCEPTION_FILTER lpAltoNivelFiltroExcecao);
```

O parâmetro *lpAltoNivelFiltroExcecao* fornece o endereço de uma função de filtro de exceção de alto nível que o Windows chamará sempre que a função *UnhandledExceptionFilter* obtiver o controle e o Windows não estiver depurando o processo. Um valor *NULL* para o parâmetro *lpAltoNivelFiltroExcecao* especifica um tratamento padrão dentro de *UnhandledExceptionFilter*.

A função de filtro tem sintaxe congruente à de *UnhandledExceptionFilter*. A função de filtro recebe um único parâmetro do tipo *LPEXCEPTION_POINTERS* e retorna um valor do tipo *LONG*. A função de filtro deve retornar um dos valores listados pela Tabela 1393.

Tabela 1393 Valores de retorno possíveis para a função de filtro.

Valor	Significado
EXCEPTION_EXECUTE_HANDLER	Retorna de *UnhandledExceptionFilter* e executa a rotina de tratamento de exceção associada. Esse valor geralmente resulta em finalização do processo.
EXCEPTION_CONTINUE_EXECUTION	Retorna de *UnhandledExceptionFilter* e continua a execução a partir do ponto da exceção. Observe que a função de filtro é livre para modificar o estado de continuação modificando as informações de exceção que seu parâmetro *LPEXCEPTION_POINTERS* fornece.
EXCEPTION_CONTINUE_SEARCH	Prossegue com a execução normal de *UnhandledExceptionFilter*, o que significa obedecer aos sinalizadores *SetErrorMod*, ou chamar a caixa de mensagem Application Error.

A função *SetUnhandledExceptionFilter* retorna o endereço da exceção anterior estabelecida com a função. Um valor de retorno *NULL* significa que não há nenhum filtro de exceção de alto nível atual. *SetUnhandledExceptionFilter* substitui o filtro de exceção de alto nível para todos os encadeamentos existentes e todos os futuros no processo chamador. O programa executa a rotina de tratamento de exceção que o parâmetro *lpAltoNivelFiltroExcecao* especifica no contexto do encadeamento que causou a falha. Como a rotina de tratamento de exceção executa dentro de um encadeamento, ela pode afetar a capacidade da rotina de tratamento de exceção de se recuperar de certas exceções, tais como uma pilha inválida.

Para compreender melhor o processamento que a função *SetUnhandledExceptionFilter* efetua, considere o programa *Handle_Exception.cpp*, contido no CD-ROM que acompanha este livro. O programa *Handle_Exception.cpp* cria um filtro de exceção não-tratada, e, depois, demonstra como o programa intercepta as exceções tratadas e as não-tratadas nos filtros de exceção. As funções *YourUnhandledExceptionFilter* e *WndProc* do programa *Handle_Exception.cpp* contêm o código que efetua o processamento operativo.

1394 TERMINANDO OS ENCADEAMENTOS

À medida que seus programas se tornarem mais complexos, algumas vezes um encadeamento deixará de completar sua execução normalmente. Como você aprendeu, seus programas precisam parar ou fazer uma pausa no encadeamento para que o sistema operacional pare de escalonar esse encadeamento para a execução. Se seus programas deixarem de completar sua execução corretamente, eles não fecharão. Em vez disso, seus programas precisarão instruir o sistema operacional a terminar o encadeamento. Para fazer isso, seus programas usarão a função *TerminateThread*. Você usará *TerminateThread* como mostrado no protótipo a seguir:

```
BOOL TerminateThread(
  HANDLE  hEncad,   // indicativo para o encadeamento
  DWORD dwCdgSaida  // código de saída para o encadeamento
);
```

O parâmetro *hEncad* identifica o encadeamento a terminar. Sob o Windows NT, o indicativo precisa ter o acesso *THREAD_TERMINATE*. O parâmetro *duCdgSaida* especifica o código de saída para o encadeamento. Use a função *GetExitCodeThread* para recuperar o valor de saída de um encadeamento. Se a função for bem-sucedida, o valor de retorno será diferente de zero. Se a função falhar, o valor de retorno será zero.

Seu programa usa *TerminateThread* para fazer um encadeamento terminar. Quando você usar *TerminateThread*, o encadeamento-alvo não terá chance de executar qualquer código do modo usuário, e seu programa não desalocará sua pilha inicial. Seu programa não notifica as bibliotecas de ligações dinâmicas (DLL) associadas que o encadeamento está terminando.

TerminateThread é uma função potencialmente perigosa que você somente deve usar nos casos mais extremos. Você deve chamar *TerminateThread* apenas se souber exatamente o que o encadeamento-alvo está fazendo, e controlar todo o código que o encadeamento-alvo poderia possivelmente estar rodando ao tempo da finalização. Por exemplo, *TerminateThread* pode resultar nos seguintes problemas:

- Se o encadeamento-alvo possuir uma seção crítica, o Windows não liberará a seção crítica.
- Se o encadeamento-alvo estiver executando certas chamadas do núcleo quando a chamada à *TerminateThread* o terminar, o estado do núcleo para o processo do encadeamento poderá ser inconsistente.
- Se o encadeamento-alvo estiver manipulando o estado global de uma biblioteca de ligações dinâmicas compartilhada, o Windows poderá destruir o estado da biblioteca, afetando outros de seus usuários.

Um encadeamento somente pode proteger a si mesmo contra *TerminateThread* controlando o acesso a seus indicativos. O indicativo do encadeamento que as funções *CreateThread* e *CreateProcess* retornam tem o acesso *THREAD_TERMINATE*, de modo que qualquer chamador que contenha um desses indicativos pode terminar seu encadeamento. Se o encadeamento-alvo for o último encadeamento de um processo quando seu programa chamar a função *TerminateThread*, o programa também terminará o processo do encadeamento. O estado do objeto encadeamento torna-se sinalizado, liberando quaisquer outros encadeamentos que estavam esperando que o encadeamento terminasse. O estado de finalização do encadeamento muda de *STILL_ACTIVE* para o valor do parâmetro *dwCdgSaida*.

Finalizar um encadeamento não remove necessariamente o objeto encadeamento do sistema. O Windows exclui um objeto de encadeamento quando seu programa fecha o último indicativo de encadeamento.

Para compreender melhor o processamento que a função *TerminateThread* executa, considere o programa *Manip_Threads*, contido no CD-ROM que acompanha este livro. O programa *Manip_Threads.cpp* permite que o usuário crie um encadeamento e depois suspenda (pressione Alt+S), reinicie (pressione Alt+R) ou termine esse encadeamento (pressione Alt+K). O programa evita as questões relacionadas com a finalização do encadeamento fazendo o encadeamento não executar atividades e somente permitindo que o usuário aceite o encadeamento criado mais recentemente. As funções *ThreadProc* e *WndProc* do programa *Manip_Thread* contêm o código que efetua o processamento operativo.

DETERMINANDO A IDENTIFICAÇÃO DE UM ENCADEAMENTO OU DE UM PROCESSO 1395

Como você aprendeu na Dica 1389, seus programas freqüentemente irão requerer um identificador temporário, ou pseudo-indicativo, para o encadeamento ou processo atual. Com menos freqüência, seus programas irão requerer um indicativo permanente ou outro valor único para representar um encadeamento ou processo em todo o sistema. A API Win32 fornece duas funções, *GetCurrentThreadId* e *GetCurrentProcessId*, que permitem que seus programas obtenham um valor *DWORD* único que o sistema operacional usa para representar o encadeamento ou o processo internamente. Você usará essas funções dentro de seus programas, como mostrado nos seguintes protótipos:

```
DWORD GetCurrentThreadId(void);
DWORD GetCurrentProcessId(void);
```

A função *GetCurrentThreadId* retorna o identificador de encadeamento do encadeamento chamador, que é o valor de retorno do encadeamento chamador. Até que o programa termine, o identificador do encadeamento identifica de forma única o encadeamento em todo o sistema.

Similarmente, a função *GetCurrentProcessId* retorna o identificador do processo do processo chamador. Esta função não tem parâmetros. O valor de retorno é o identificador do processo do processo chamador. Até que o processo termine, o identificador do processo identifica unicamente o processo em todo o sistema.

O programa *ShowCurrent* que a Dica 1389 apresentou usa ambas as funções *GetCurrentProcessID* e *GetCurrentThreadID*.

Nota: Similar a um indicativo, a DWORD que GetCurrentProcessID ou GetCurrentThreadID retorna é um valor único que identifica o encadeamento ou o processo no sistema operacional. Não confunda o Identificador do Processo e o Identificador do Encadeamento com os pseudo-indicativos que GetCurrentProcess e GetCurrentThread retornam.

1396 COMPREENDENDO COMO O SISTEMA OPERACIONAL ESCALONA OS ENCADEAMENTOS

Como você aprendeu, o Win32 é um sistema operacional multiencadeado. Ele é capaz de tratar um grande número de encadeamentos ou processos subseqüentes em ordem. No entanto, como as dicas anteriores indicaram, ele tratará certos encadeamentos mais rapidamente, ou em uma ordem diferente que tratará outros Por exemplo, o sistema operacional Win32 tenderá a dar maior prioridade aos encadeamentos no processo atual do que aos processos que estiverem em execução no segundo plano.

Na verdade, o sistema operacional Win32 escalona todo encadeamento que solicita processamento da CPU (isto é, todos os encadeamentos ativos) com base em seus níveis de prioridade. A dica a seguir explicará os *níveis de prioridade em detalhes. Quando o sistema atribui uma CPU a um encadeamento, ele trata todos os encadeamentos* da mesma prioridade como iguais. Em outras palavras, o sistema atribui o primeiro encadeamento na fila com o nível de prioridade 31 para uma CPU. Portanto, é importante observar que se você sempre tem pelo menos um encadeamento de prioridade 31 para cada CPU, os encadeamentos com prioridades menores nunca serão executados. Os programadores chamam essa condição de prioridade de *starvation* (fome). Starvation ocorre quando alguns encadeamentos usam tanto tempo da CPU que nenhum outro encadeamento consegue entrar em execução.

Quando o sistema completa os encadeamentos de prioridade 31, ele começa a atribuir os encadeamentos de prioridade 30. Pode parecer, então, que os encadeamentos de baixa prioridade nunca serão executados (como regra) em um sistema assim Ocorre, no entanto, que até os encadeamentos de prioridade mais alta, freqüentemente não requerem tempo de CPU, o que libera a CPU para tratar os encadeamentos de baixa prioridade.

Finalmente, você deve compreender que, se um encadeamento de baixa prioridade estiver em execução — mesmo se ele estiver no meio de sua fatia de tempo — e o sistema determinar que um encadeamento de alta prioridade está aguardando para entrar em execução, o sistema parará imediatamente a execução do encadeamento de baixa prioridade e iniciará a execução do encadeamento de prioridade mais alta. O encadeamento de prioridade mais alta sempre interromperá o de prioridade mais baixa, independentemente do que o encadeamento de baixa prioridade estiver fazendo ou qual estado ele estiver em sua execução.

Nota: A Microsoft se reserva o direito de modificar o algoritmo que os sistemas operacionais baseados no Win32 usam para o escalonamento dos encadeamentos — o Windows 95, o Windows NT 3.51 e o NT 4.0 têm todos diferentes variações do algoritmo de escalonamento. Embora você deva compreender como a prioridade do encadeamento funciona e deva usá-la com cuidado, seus programas não devem estar tão amarrados ao método que um sistema operacional usa para gerenciar a prioridade dos encadeamentos, pois isso causa problemas no caso de a Microsoft alterar o algoritmo de escalonamento em versões futuras.

1397 APRESENTANDO OS NÍVEIS DE PRIORIDADE

Como visto na dica anterior, o sistema operacional Win32 escalona todos os encadeamentos ativos com base em seus níveis de prioridades atuais. Os níveis de prioridade vão de 0 (o menor) até 31 (o maior). O sistema operacional atribui o nível 0 a um encadeamento especial do sistema chamado *encadeamento da página zero*. Esse encadeamento é responsável por zerar quaisquer páginas livres no sistema quando não existirem outros encadeamentos que precisem executar trabalho no sistema. Não é possível para outro encadeamento qualquer, além do encadeamento página zero, ter um nível de prioridade zero.

Quando você cria encadeamentos, não usa números para lhes atribuir níveis de prioridade. Em vez disso, o sistema usa um processo em duas etapas para determinar o nível de prioridade do encadeamento. O primeiro passo é atribuir uma classe de prioridade a um processo. A classe de prioridade do processo informa ao sistema a prioridade que o processo requer comparada com os outros processos em execução. A dica a seguir explica as classes de prioridade em detalhes. O segundo passo é atribuir um nível de prioridade relativo a cada encadeamento que o processo vier a possuir.

Assim que você cria um encadeamento dentro de um processo, o seu nível de prioridade é o mesmo que o nível de prioridade do processo. Na Dica 1400, logo à frente você aprenderá como usar a API Win32 para alterar o nível de prioridade relativo de um encadeamento.

COMPREENDENDO AS CLASSES DE PRIORIDADE DO WINDOWS 1398

Como explicou a dica anterior, o Windows usa um processo de duas etapas para determinar a prioridade de um encadeamento. O primeiro passo é determinar a classe de prioridade do processo O Win32 suporta quatro classes diferentes de prioridade: ocioso, normal, alta e tempo real. A Tabela 1398 detalha as classes de prioridade.

Tabela 1398 Classes de prioridade possíveis.

Prioridade	Significado
HIGH_PRIORITY_CLASS	Indica um processo que executa tarefas críticas que o sistema operacional precisa executar imediatamente para que o processo funcione corretamente. Os encadeamentos de um processo de classe de alta prioridade interrompem os encadeamentos de processos de classe normal ou de prioridade ociosa Um exemplo é a Lista de Tarefas do Windows, que precisa responder rapidamente quando o usuário chama, independentemente da carga sobre o sistema operacional. Tenha extremo cuidado ao usar a classe de alta prioridade, pois um aplicativo de classe de alta prioridade poderá utilizar praticamente todos os ciclos disponíveis. O nível de prioridade de um processo *HIGH_PRIORITY_CLASS* é 13.
IDLE_PRIORITY_CLASS	Indica um processo cujos encadeamentos rodam somente quando o sistema está ocioso e os encadeamentos de qualquer processo em execução em uma classe de prioridade mais alta os interrompem. Um exemplo é um protetor de tela. Os processos-filho herdam a classe de prioridade ociosa. O nível de prioridade de *IDLE_PRIORITY_CLASS* é 6
NORMAL_PRIORITY_CLASS	Indica um processo normal sem necessidades de escalonamento especiais. O nível de prioridade de *NORMAL_PRIORITY_CLASS* é 8.
REALTIME PRIORITY CLASS	Indica um processo que tem a prioridade mais alta possível. Os encadeamentos de um processo de classe de prioridade de tempo real interrompem os encadeamentos de todos os outros processos, incluindo os processos do sistema operacional que executam tarefas importantes. Por exemplo, um processo de tempo real que executa por mais do que um intervalo muito breve pode fazer com que os caches não descarreguem seus dados no disco ou que o mouse não responda. O nível de prioridade de *REALTIME_PRIORITY_CLASS* é 24.

É muito útil que você compreenda mais completamente o impacto dos diferentes níveis de classe. Por exemplo, apenas use a definição *HIGH_PRIORITY_CLASS* quando absolutamente necessário. O processo mais comum para usar a definição *HIGH_PRIORITY_CLASS* é o Windows *Explorer*. Embora a maioria dos encadeamentos adormeça durante a execução normal, os usuários esperam que o computador responda quando eles quiserem acessá-lo. Portanto, o Windows dá aos encadeamentos do Explorer a prioridade mais alta, e interromperá quase qualquer outro quando o usuário selecionar uma opção do computador Se você criar programas que também usem a definição *HIGH_PRIORITY_CLASS*, o computador poderá não responder de forma adequada e o Windows poderá até mesmo travá-lo.

Geralmente, seus programas usarão *IDLE_PRIORITY_CLASS* para aplicativos de monitoração do sistema. Por exemplo, você poderia querer escrever um aplicativo que exibisse periodicamente a quantidade de RAM livre no sistema. Como você não irá querer que o aplicativo interfira com o desempenho de outra tarefa mais crítica, você definirá a classe do processo periódico como *IDLE_PRIORITY_CLASS*.

O Windows atribui automaticamente a definição *NORMAL_PRIORITY_CLASS* a qualquer processo ao qual você não atribui explicitamente outra definição. Seus programas geralmente devem usar a definição *NORMAL_PRIORITY_CLASS*. Observe que, quando o usuário traz um processo para o primeiro plano, o sistema

operacional aumenta a prioridade relativa do processo para proporcionar melhor velocidade de execução. Por exemplo, no Windows 95, o sistema operacional soma um ao contador de prioridade do processo de primeiro plano.

Em geral, seus programas nunca devem usar a definição *REALTIME_PRIORITY_CLASS* porque a prioridade de tempo real é uma prioridade muito alta — na verdade, é ainda mais alta, que o gerenciamento de encadeamentos do sistema operacional. Os encadeamentos no sistema que controlam o mouse e o teclado, descarga no disco em segundo plano, até a interceptação de Ctrl+Alt+Del, todos executam mais em reduzida do que a prioridade de tempo real. Os programas que usam prioridade de tempo real freqüentemente terão efeitos adversos significativos no sistema do usuário.

1399 ALTERANDO A CLASSE DE PRIORIDADE DE UM PROCESSO

O sistema operacional Win32 atribui prioridade normal a todo processo novo. No entanto, freqüentemente, seus programas podem requerer modificações na classe de prioridade de um processo atual. Você pode usar as funções *GetPriorityClass* e *SetPriorityClass* para gerenciar uma classe de prioridade de um processo. A função *GetPriorityClass* retorna a classe de prioridade para os processos específicos. Você usará a função *GetPriorityClass* dentro de seus programas como mostrado no protótipo a seguir:

```
DWORD  GetPriorityClass(HANDLE hProcesso);
```

O indicativo do processo *hProcesso* identifica o processo. Sob o Windows NT, o indicativo *hProcesso* precisa ter o direito de acesso *PROCESS_QUERY_INFORMATION*. Se a função *GetPriorityClass* for bem-sucedida, o valor de retorno será a classe de prioridade do processo especificado. Se falhar, o valor de retorno será 0. A classe de prioridade do processo especificado é um dos valores já listados na Tabela 1398.

Similarmente, a função *SetPriorityClass* define a classe de prioridade para o processo especificado. A classe de prioridade, juntamente com o valor de prioridade de cada encadeamento de processo determina o nível de prioridade básica de cada encadeamento. Você usará a função *SetPriorityClass* dentro de seus programas de acordo com o protótipo a seguir:

```
BOOL SetPriorityClass(
   HANDLE hProcesso,    // indicativo para o processo
   DWORD dwClassePrior  // valor da classe de prioridade
);
```

Exatamente como com a função *GetPriorityClass*, o indicativo *hProcesso* identifica o processo. Sob o Windows NT, o indicativo *hProcesso* precisa ter o direito de acesso *PROCESS_SET_INFORMATION*. O parâmetro *dwClassePrior* especifica a classe de prioridade para o processo. O parâmetro *dwClassePrior* pode ser qualquer um dos valores já listados na Tabela 1398. Se a função *SetThreadPriority* for bem-sucedida, o valor de retorno será diferente de zero. Se ela falhar, o valor de retorno será 0.

Todo encadeamento tem um nível de prioridade básico, que o Windows determina com base no valor de prioridade do encadeamento e da classe de prioridade do processo do encadeamento. O sistema usa o nível de prioridade base de todos os encadeamentos executáveis para determinar qual encadeamento receberá a próxima fatia de tempo da CPU. A função *SetThreadPriority* lhe permite definir a prioridade base de um encadeamento relativo à classe de prioridade de seu processo. A dica a seguir usa *SetThreadPriorityFunction* para definir o nível de prioridade de um encadeamento.

O CD-ROM que acompanha este livro inclui o programa *Get_Set_Priority*, que permite que o usuário selecione a classe de prioridade do processo. Após cada seleção, o programa executará uma função que utiliza intensamente a CPU e exibirá os resultados, juntamente com a classe de prioridade retornada pelo sistema operacional.

1400 DEFININDO A PRIORIDADE RELATIVA DE UM ENCADEAMENTO

Como você aprendeu, o Windows define um nível de prioridade de um encadeamento com base na classe de processo de um encadeamento e o seu nível de prioridade. Na dica anterior, você aprendeu como usar a função *SetProcessClass* para modificar a classe de prioridade de um processo. Para alterar o nível de prioridade dos en-

cadeamentos dentro de um processo, seus programas devem usar a função *SetThreadPriority*. A função *SetThreadPriority* define o valor de prioridade para o encadeamento especificado. Este valor, juntamente com a classe de prioridade do processo do encadeamento, determina o nível de prioridade de base do encadeamento. Você usará a função *SetThreadPriority* dentro de seus programas, como segue:

```
BOOL SetThreadPriority(
   HANDLE hEncade,   // indicativo para o encadeamento
   int nPrioridade   // nível de prioridade do encadeamento
);
```

O parâmetro *hEncad* identifica o encadeamento cujo valor de prioridade a função definirá. Sob o Windows NT, o indicativo precisa ter o direito de acesso *THREAD_SET_INFORMATION*. O parâmetro *nPrioridade* especifica o valor de prioridade para o encadeamento.

Esse parâmetro pode ser um dos valores listados na tabela 1400.

Tabela 1400 Os valores do nível de prioridade do encadeamento para o parâmetro ***nPrioridade***.

Prioridade	Significado
THREAD_PRIORITY_ABOVE_NORMAL	Indica 1 ponto acima da prioridade normal para a classe de prioridade.
THREAD_PRIORITY_BELOW_NORMAL	Indica 1 ponto abaixo da prioridade normal para a classe de prioridade.
THREAD_PRIORITY_HIGHEST	Indica 2 pontos acima de prioridade normal para a classe de prioridade.
THREAD_PRIORITY_IDLE	Indica um nível de prioridade de base de 1 para *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS*, ou *HIGH_PRIORITY_CLASS* e um nível de prioridade de 16 para os processos *REALTIME_PRIORITY_CLASS*.
THREAD_PRIORITY_LOWEST	Indica 2 pontos abaixo da prioridade normal para a classe de prioridade.
THREAD_PRIORITY_NORMAL	Indica prioridade normal para a classe de prioridade.
THREAD_PRIORITY_TIME_CRITICAL	Indica um nível de prioridade de base de 15 para *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS*, ou *HIGH_PRIORITY_CLASS* e um nível de prioridade de base de 31 para os processos *REALTIME_PRIORITY_CLASS*.

Se a função *SetThreadPriority* for bem-sucedida, o valor de retorno será diferente de zero. Se ela falhar, o valor de retorno será zero. Como você aprendeu, todo encadeamento tem um nível de prioridade de base determinado pelo valor da sua prioridade e pela classe da prioridade de seu processo. O sistema usa o nível de prioridade de base de todos os encadeamentos para determinar qual encadeamento recebe a próxima fatia do tempo da CPU. O sistema escalona os encadeamentos de uma forma de duas vias em cada nível de prioridade, e, somente quando não existirem encadeamentos executáveis em um nível mais alto, o sistema escalonará encadeamentos em um nível mais baixo.

A função *SetThreadPriority* lhe permite definir o nível de prioridade de base do encadeamento relativo à classe de prioridade de seu processo. Por exemplo, especificar *THREAD_PRIORITY_HIGHEST* em uma chamada à *SetThreadPriority* de um processo *IDLE_PRIORITY_CLASS* define o nível de prioridade de base para 6. Para os processos *IDLE_PRIORITY_CLASS*, *NORMAL_PRIORITY_CLASS* e *HIGH_PRIORITY_CLASS*, o sistema dinamicamente aumenta o nível de prioridade de base de um encadeamento, quando os eventos ocorrem, que são importantes para o encadeamento (tais como outro encadeamento indo para um estado de ocioso). Os processos *REALTIME_PRIORITY_CLASS* não recebem aumentos dinâmicos. Todos os encadeamentos iniciam em *THREAD_PRIORITY_NORMAL*.

Use a classe de prioridade de um processo para diferenciar entre os aplicativos que dependem criticamente do tempo e aqueles que têm requisitos de escalonamento normais ou abaixo do normal. Use valores de prioridade para diferenciar entre as prioridades relativas das tarefas de um processo. Por exemplo, um encadeamento que trata entrada para uma janela poderia ter um nível de prioridade mais alto que um encadeamento que realiza cálculos que utilizam muito a CPU.

Ao manipular prioridades, seja muito cuidadoso para garantir que um encadeamento de alta prioridade não consuma todo o tempo disponível da CPU. Um encadeamento com o nível de prioridade de base acima de 11 interfe com a operação normal do sistema operacional. Usar *REALTIME_PRIORITY_CLASS* pode fazer com que os caches de disco não sejam gravados no disco, que o mouse não seja responsivo e assim por diante.

1401 OBTENDO O NÍVEL DE PRIORIDADE ATUAL DE UM ENCADEAMENTO

Como você aprendeu na dica anterior, seus programas podem usar a função *SetThreadPriority* para alterar o nível de prioridade atual de um encadeamento. Normalmente, seus programas irão requerer informações sobre o nível de prioridade atua. de um encadeamento, em geral como o passo antes de uma chamada à *SetThreadPriority*. A função *GetThreadPriority* retorna o valor de prioridade para o encadeamento especificado. Este valor, juntamente com a classe de prioridade do processo do encadeamento, determina o nível de prioridade de base do encadeamento. Você usará a função *GetThreadPriority* dentro de seus programas, como mostra o protótipo a seguir:

```
int GetThreadPriority(
   HANDLE hEncad     // indicativo para encadeamento
);
```

Como com a função *SetThreadPriority*, o parâmetro *hEncad* identifica o encadeamento. Se for bem-sucedida, a função retornará *THREAD_PRIORITY_ERROR_RETURN*. Para obter informações ampliadas sobre o erro, chame *GetLastError*. O nível de prioridade do encadeamento é um dos valores já detalhados na Tabela 1400.

Lembre-se, todo encadeamento tem um nível de prioridade base que o sistema operacional determina pelo valor de prioridade do encadeamento e a classe de prioridade do processo do encadeamento. O sistema operacional usa o nível de prioridade base de todos os encadeamentos executáveis para determinar qual encadeamento receberá a próxima fatia de tempo da CPU. O sistema operacional escalona os encadeamentos de uma forma de duas vias no nível de cada prioridade, e, somente quando não existem encadeamentos executáveis em um nível mais alto o sistema operacional escalona os encadeamentos em um nível mais baixo.

Nota: Sob o Windows NT, o indicativo precisa ter o acesso THREAD_QUERY_INFORMATION.

1402 OBTENDO O CONTEXTO DE UM ENCADEAMENTO

O Windows armazena informações sobre cada encadeamento dentro da estrutura *CONTEXT*. À medida que *seus programas manipularem os encadeamentos mais e mais frequentemente, eles poderão requerer informações* sobre o contexto de um encadeamento. A função *GetThreadContext* recupera o contexto do encadeamento especificado. Você usará a função *GetThreadContext* como mostrado no protótipo a seguir:

```
BOOL GetThreadContext (
   HANDLE hEncad,          // indicativo do encadeamento com contexto
   LPCONTEXT lpContexto    // endereço da estrutura do // contexto
);
```

O parâmetro *hEncad* identifica um indicativo aberto de um encadeamento cujo contexto a função deverá recuperar. O parâmetro *lpContexto* aponta para o endereço de uma estrutura *CONTEXT* que recebe o contexto apropriado do encadeamento especificado. O valor do membro *ContextFlags* dessa estrutura especifica quais porções do contexto de um encadeamento são recuperadas. A estrutura *CONTEXT* é altamente específica do computador. Atualmente, existem estruturas *CONTEXT* definidas para os processadores da Intel, MIPS, Alpha e PowerPC.

Você deve usar a função *GetThreadContext* para recuperar o contexto de um encadeamento especificado. A função permite que seus programas recuperem um contexto seletivo com base no valor do membro *ContextFlags*

da estrutura *CONTEXT*. O indicativo do encadeamento que o parâmetro *hEncad* identifica está tipicamente sendo depurado, mas a função também poderá operar quando não estiver sendo depurada. Você não pode obter um contexto válido para um encadeamento em execução. Você precisa usar a função *SuspendThread* para suspender o encadeamento antes de chamar *GetThreadContext*.

Nota: Sob o Windows NT, o indicativo precisa ter o acesso THREAD_GET_CONTEXT ao encadeamento.

Fazendo uma Pausa e Reiniciando os Encadeamentos 1403

Em dicas anteriores, você aprendeu que seus programas podem operar os encadeamentos em um estado suspenso (usando o sinalizador *CREATE_SUSPENDED* com a função *CreateProcess* ou *CreateThread*). Quando você cria um encadeamento suspenso, o sistema cria o objeto núcleo que identifica o encadeamento, cria a pilha do encadeamento e inicializa os membros de registradores da CPU dentro da estrutura *CONTEXT*. No entanto, a função criadora dá ao objeto encadeamento um contador de suspensão inicial de 1, o que significa que o sistema nunca atribuirá tempo de CPU para executar o encadeamento. Para permitir que o encadeamento inicie sua execução, outro encadeamento precisa chamar a função *ResumeThread* e passá-la para o indicativo para o encadeamento suspenso. A função *ResumeThread* decrementa o contador de suspensão de um encadeamento. Quando *ResumeThread* decrementa o contador de suspensão para zero, o programa reinicia a execução do encadeamento. Você usará a função *ResumeThread* dentro de seus programas, como segue:

```
DWORD ResumeThread(HANDLE hEncad);
```

O parâmetro *hEncad* especifica um indicativo para o encadeamento que o programa reiniciará. Você pode suspender um encadeamento múltiplas vezes; no entanto, precisa chamar *ResumeThread* o mesmo número de vezes para que ele retome a execução.

A função *ResumeThread* verifica o contador de suspensão do encadeamento em questão. Se o contador for zero, o encadeamento não estará atualmente suspenso. Caso contrário, a função *ResumeThread* decrementa o contador de suspensão do encadeamento. Se o valor resultante for zero, então o programa retomará a execução do encadeamento. Se o valor de retorno for zero, você não suspendeu o encadeamento. Se o valor de retorno for 1, você suspendeu o encadeamento, mas o programa o reiniciou. Se o valor de retorno for maior que 1, o encadeamento ainda estará suspenso.

Observe que, enquanto informam os eventos de depuração, todos os encadeamentos dentro do processo que está informando estão congelados. O sistema operacional espera que os depuradores usem as funções *SuspendThread* e *ResumeThread* para limitar o conjunto de encadeamentos que podem executar dentro de um processo. É possível "executar passo a passo" um único encadeamento suspendendo todos os encadeamentos em um processo — exceto aquele que está informando um evento de depuração. Os outros encadeamentos não são liberados por uma operação contínua se estiverem suspensos.

Nota: Sob o Windows NT, o indicativo hProcesso precisa ter acesso THREAD_SUSPEND_RESUME ao processo.

Compreendendo a Sincronização de Encadeamentos 1404

Você já viu que o Windows suporta a execução de múltiplos encadeamentos. Em um ambiente onde um ou mais encadeamentos podem executar concorrentemente, com frequência torna-se importante permitir que seus programas sincronizem as atividades dos vários encadeamentos. O sistema operacional Win32 fornece vários objetos de sincronização que permitem que os encadeamentos sincronizem suas ações com outros encadeamentos. Na dica a seguir, você aprenderá mais sobre os objetos de sincronização específicos.

Em geral, um encadeamento sincroniza-se com outro "colocando-se para dormir". Quando o encadeamento adormece, o sistema operacional não escalona mais o tempo da CPU para ele, e, portanto, o encadeamento pára de executar. No entanto, logo antes de adormecer, o encadeamento diz ao sistema operacional qual "evento especial" (tal como uma digitação, um clique no mouse e uma finalização de algoritmos) precisa ocorrer para que o encadeamento comece a executar novamente.

O sistema operacional, por sua vez, continua ciente da solicitação do encadeamento e atento para ver se o evento especial ocorre. Quando o evento ocorre, o sistema operacional alerta o encadeamento, que, então, se torna elegível novamente para ser escalonado para acessar a CPU. Em algum momento, a CPU escalonará o en-

cadeamento e continuará a execução do escalonamento — o que significa que o encadeamento está agora *sincronizado* com a ocorrência do evento especial.

1405 DEFININDO OS CINCO PRINCIPAIS OBJETOS DE SINCRONIZAÇÃO

O Windows suporta diferentes tipos de objetos de sincronização. Desses tipos, os cinco mais comumente usados são *seções críticas, mutexes, semáforos* e *temporizadores* aguardáveis. Dicas posteriores discutirão alguns desses tipos em detalhes. No entanto, é útil compreender as definições simples de cada tipo. A Tabela 1405 lista os cinco principais tipos de objetos de sincronização e suas ações.

Tabela 1405 Os cinco principais tipos de sincronização de encadeamentos.

Tipo	Uso e Ações
Seção crítica	Uma seção crítica é uma seção pequena de código que requer acesso exclusivo para certos dados compartilhados antes que o código possa executar. No entanto, você *somente pode usar seções críticas para sincronizar encadeamentos dentro de um único* processo.
Mutexes	Os mutexes são muito parecidos com as seções críticas. No entanto, seus programas usam mutexes para sincronizar acesso de dados entre múltiplos processos. Além disso, os mutexes são *objetos do núcleo*, o que significa que seus programas, na verdade, criarão um mutex usando uma função da API, tal como *CreateMutex*.
Semáforos	Seus programas usarão objetos semáforos para contar recursos. Um único encadeamento pode usar um semáforo para contar o número de recursos disponíveis e alocar recursos. Por exemplo, se um computador tem três portas seriais, você pode criar um semáforo com um contador de recursos de 3. Toda vez que um encadeamento acessar uma porta serial, o contador de recursos do semáforo é incrementado em 1, e, toda vez que um encadeamento libera uma porta serial, o contador de recursos é incrementado em 1. Portanto, os encadeamentos podem chamar o semáforo e aguardar até que ele se torne disponível antes de tentar acessar as portas seriais. Ao contrário dos mutexes e das seções críticas, os semáforos *não são de propriedade de encadeamentos*.
Eventos	Os objetos eventos são a forma mais primitiva de objetos de sincronização e são bem diferentes dos mutexes e dos semáforos. Tipicamente, seus programas usarão mutexes e semáforos para controlar o acesso aos dados ou recursos Por outro lado, seus programas usarão eventos para sinalizar o término de uma operação. Seus programas mais freqüentemente usarão eventos para iniciar um segundo encadeamento após o primeiro terminar algumas partes de seu processamento.
Temporizadores aguardáveis	Um temporizador aguardável é um objeto do núcleo que periodicamente sinaliza a si mesmo, ou em um tempo específico ou em intervalos regulares. Você pode pensar no temporizador aguardável como um despertador interno para seus programas. Por exemplo, você poderia escrever um programa de escalonamento que alertasse o usuário toda hora dos novos compromissos naquela hora. Em vez de percorrer em um laço constantemente e aguardar que a hora mude, seu programa poderia criar um temporizador aguardável que sinalizasse o programa da mudança de hora. Os temporizadores aguardáveis existem somente no Windows NT 4 e superiores. O Windows 95 não suporta os temporizadores aguardáveis.

1406 CRIANDO UMA SEÇÃO CRÍTICA

Como você aprendeu na dica anterior, o tipo mais simples de sincronização de encadeamento a usar é uma seção crítica. Como você também aprendeu, uma seção crítica permite que seus programas controlem o acesso a um determinado dado ou a uma função dentro do programa, garantindo que somente um encadeamento de cada

vez acesse esses dados ou que todos os outros encadeamentos internos ao processo tenham completado seu processamento antes de a seção crítica executar.

Criar uma seção crítica é relativamente fácil. Primeiro, seu programa precisa alocar uma estrutura de dados *CRITICAL_SECTION* dentro do processo em execução. Seu programa precisa alocar a estrutura de dados *CRITICAL_SECTION* globalmente para que diferentes encadeamentos dentro do processo atual possam acessar a ocorrência *CRITICAL_SECTION* do programa. Normalmente, isso significa que sua ocorrência *CRITICAL_SECTION* será uma variável global.

Após seu programa alocar a estrutura de dados *CRITICAL_SECTION*, ele precisará seguir dois passos para criar e enttar na seção crítica. Seu programa precisará primeiro chamar a função *InitializeCriticalSection* para inicializar a seção e depois seu programa precisará chamar a função *EnterCriticalSection* quando estiver preparado para entrar na seção crítica. A função *EnterCriticalSection* espera para o encadeamento possuir o objeto de seção crítica especificado. A função retorna quando o sistema operacional concede posse ao encadeamento chamador. Seus programas usarão a função *EnterCriticalSection*, como mostra o protótipo a seguir:

```
void EnterCriticalSection(LPCRITICAL_SECTION
                          lpSecaoCritica);
```

O parâmetro *lpSecaoCritica* aponta para o objeto de seção crítica. Para habilitar o acesso mutuamente exclusivo a um recurso compartilhado, cada encadeamento chama a função *EnterCriticalSection* ou *TryEnterCriticalSection* para solicitar a posse da seção crítica antes de executar qualquer seção de código que acesse o recurso protegido. A diferença é que a função *TryEnterCriticalSection* retorna de forma imediata, independentemente se obteve posse da seção crítica, enquanto a função *EnterCriticalSection* bloqueia o processamento até que o encadeamento possa tomar posse da seção crítica. Quando tiver terminado de executar o código protegido, o encadeamento usará a função *LeaveCriticalSection* para abrir mão da posse, permitindo que outro encadeamento se torne o possuidor e acesse o recurso protegido. O encadeamento controlador precisará chamar *LeaveCriticalSection* uma vez sempre que entrar na seção crítica. O encadeamento entrará na seção crítica toda vez que *EnterCriticalSection* ou *TryEnterCriticalSection* forem bem-sucedidas.

Após um encadeamento possuir uma seção crítica, ele poderá fazer chamadas adicionais à função *EnterCriticalSection* ou *TryEnterCriticalSection* sem bloquear sua própria execução. Isso evita que um encadeamento crie um impasse (*deadlock*) consigo mesmo, isto é, pare sua própria execução enquanto aguarda uma seção crítica que ele já possui.

Qualquer encadeamento de processo pode usar a função *DeleteCriticalSection* para liberar os recursos do sistema que o programa alocou quando inicializou o objeto de seção crítica. Após o encadeamento ter chamado a função *DeleteCriticalSection*, o programa não poderá mais usar o objeto seção crítica para sincronização.

Nota: Embora os membros da estrutura de dados CRITICAL_SECTION estejam definidos dentro do arquivo de cabeçalho winbase.h, seus programas não devem tentar acessar os membros da estrutura porque o Windows gerencia essas informações internamente, e modificações feitas aos membros podem causar erros fatais de sistema.

USANDO UMA SEÇÃO CRÍTICA SIMPLES 1407

Foi visto na dica anterior que criar e usar uma seção crítica é, no mínimo, um processo de três etapas. Você precisa criar a seção, inicializá-la e entrar nela. Antes de você poder sincronizar os encadeamentos com uma seção crítica, é preciso inicializar a seção crítica, passando o endereço da estrutura de dados *CRITICAL_SECTION* como o único parâmetro. Quando você chegar ao início da seção crítica, seu programa precisará chamar a função *EnterCriticalSection* ou *TryEnterCriticalSection*, novamente passando o endereço da estrutura de dados *CRITICAL_SECTION* como o único parâmetro. Após os encadeamentos terem sincronizados, a seção crítica permanecerá até que seu programa saia dela ou a elimine.

Para compreender melhor o processamento que seus programas executarão quando você gerenciar as seções críticas, considere o programa *Crit_Section.cpp*, contido no CD-ROM que acompanha este livro. O programa usa uma seção crítica, que contém uma instrução de adormecimento de cinco segundos para permitir que somente um encadeamento de cada vez execute o código crítico. Toda vez que o usuário seleciona a opção *Testar!*, o programa cria outro encadeamento que, por sua vez, aguarda por seu acesso à seção crítica. O código operativo do programa *Crit_Section.cpp* está dentro das funções *ChildThreadProc* e *WndProc*.

1408 USANDO WAITFORSINGLEOBJECT PARA SINCRONIZAR DOIS ENCADEAMENTOS

Como você aprendeu, muitas atividades de sincronização girarão em torno de aguardar um ou mais encadeamentos antes de continuar o processamento do encadeamento atual. Quando seu encadeamento atual estiver aguardando a CPU retornar de outro encadeamento, seus programas deverão usar a função *WaitForSingleObject*, que retornará quando um dos seguintes eventos ocorrerem:

- O objeto especificado estiver no estado sinalizado, e
- O intervalo de tempo-limite transcorrer.

Seus programas usarão a função WaitForSingleObject como mostrado no seguinte protótipo:

```
DWORD WaitForSingleObject(
   HANDLE  hIndic,     // indicativo do objeto a aguardar
   DWORD   dwMilisseg  // intervalo de tempo limite em ms
);
```

A função *WaitForSingleObject* aceita dois parâmetros, *hIndic* e *dwMilisseg*. O parâmetro *hIndic* identifica o objeto para o qual a função deve aguardar. O parâmetro *dwMilisseg* especifica o intervalo de tempo-limite em milissegundos. A função *WaitForSingleObject* retornará se o intervalo transcorrer, mesmo se o estado do objeto for não-sinalizado. Se *dwMilisseg* for zero, a função testará o estado do objeto e retornará imediatamente. Se *dwMilisseg* for *INFINITE*, o intervalo de tempo-limite da função nunca transcorrerá. A Tabela 1408.1 contém a lista dos tipos de objeto cujos indicativos você pode especificar (isto é, tipos de objeto os quais a função *WaitForSingleObject* pode aguardar).

Tabela 1408.1 Os objetos para os quais ***WaitForSingleObject*** pode esperar.

Tipo de Objeto	Descrição
Mudança de notificação	A função *FindFirstChangeNotification* retorna o indicativo. O estado do objeto *mudança de notificação* é sinalizado quando um tipo especificado de alteração ocorre dentro de um diretório ou árvore de diretório que você especifica.
Entrada no console	A função *CreateFile* ou *GetStdHandle* retorna o indicativo quando você especifica o valor *CONIN$*. O estado do objeto entrada no console é sinalizado quando há uma entrada não-lida no buffer de entrada do console e será não-sinalizada quando o buffer de entrada estiver vazio.
Evento	A função *CreateEvent* ou *OpenEvent* retorna o indicativo. Ambas as funções *SetEvent* e *PulseEvent* definem explicitamente o estado de um objeto *evento* como sinalizado. Seus programas precisam usar a função *ResetEvent* para reinicializar manualmente o estado de cada objeto evento como não-sinalizado. Para um objeto evento de auto-reinicialização, a função de espera reinicializa o estado do objeto como não-sinalizado antes de retornar. Você também pode usar os objetos evento em operações sobrepostas, nas quais o sistema define o estado.
Mutex	A função *CreateMutex* ou *OpenMutex* retorna o indicativo. O estado de um objeto mutex é sinalizado quando nenhum encadeamento o possui. A função de espera solicitará a posse do mutex para o encadeamento chamador, modificando o estado do mutex para não-sinalizado quando o sistema operacional conceder a posse ao mutex.
Processo	A função *CreateProcess* ou *OpenProcess* retorna o indicativo. O estado de um objeto processo é sinalizado quando o processo termina.
Semáforo	A função *CreateSemaphore* ou *OpenSemaphore* retorna o indicativo. Um objeto *semáforo* mantém um contador entre zero, e algum valor máximo. Seu estado será sinalizado quando seu contador for maior que zero e não-sinalizado quando seu contador for zero. Se o estado atual for sinalizado, a função de espera decrementará o contador em 1.

Tabela 1408.1 Os objetos os quais ***WaitForSingleObject*** pode esperar (*Continuação*)

Tipo de Objeto	Descrição
Encadeamento	A função *CreateProcess*, *CreateThread* ou *CreateRemoteThread* retorna o indicativo. O estado de um objeto encadeamento é sinalizado quando o encadeamento termina.
Temporizador	A função *CreateWaitableTimer* ou *OpenWaitableTimer* retorna o indicativo. Ative o temporizador chamando a função *SetWaitableTimer*. O estado de um temporizador ativo é sinalizado quando ele atinge seu tempo devido. Você pode desativar o temporizador chamando a função *CancelWaitableTimer*

Nota: No Windows NT, o indicativo precisa ter o direito de acesso SYNCHRONIZE.

Se a função *WaitForSingleObject* falhar, o valor de retorno será *WAIT_FAILED*. Se a função *WaitForSingleObject* for bem-sucedida, o valor de retorno indicará o evento que fez a função retornar. O valor de retorno para a invocação bem-sucedida da função é qualquer um dos valores detalhados na Tabela 1408.2.

Tabela 1408.2 Possíveis valores de retorno para a função ***WaitForSingleObject*** em caso de sucesso.

Valor	Significado
WAIT_ABANDONED	O objeto especificado é um objeto mutex que o encadeamento que possuiu o objeto mutex não liberou antes de terminar. O sistema operacional concede a posse do objeto mutex ao encadeamento chamador e define o mutex como não-sinalizado.
WAIT_OBJECT_0	O estado do objeto especificado é sinalizado.
WAIT_TIMEOUT	O intervalo de tempo-limite transcorreu e o estado do objeto é não-sinalizado.

A função *WaitForSingleObject* verifica o estado atual do objeto especificado. Se o estado do objeto for não-sinalizado, o encadeamento chamador entra em um estado de espera eficiente. O encadeamento consome muito pouco tempo de processador enquanto espera que o estado do objeto se torne sinalizado ou que o intervalo de tempo-limite transcorra. Antes de retornar, uma função de espera modifica o estado de alguns tipos de objetos de sincronização. A modificação ocorre somente para o objeto ou os objetos cujos estados sinalizados fizeram a função retornar. Por exemplo, a função de espera decrementa o contador de um objeto *semáforo* por um incremento.

Você precisa ser cuidadoso ao usar as funções de espera e o intercâmbio dinâmico de dados. Se um encadeamento criar quaisquer janelas, ele precisará processar mensagens. O intercâmbio dinâmico de dados envia mensagens para todas as janelas no sistema. Se você tiver um encadeamento que usa uma função de espera com nenhum intervalo de tempo-limite, o sistema cairá em um impasse (*deadlock*). Portanto, se você tiver um encadeamento que cria janelas, use as funções *MsgWaitForMultipleObjects* ou *MsgWaitForMultipleObjectsEx*, em vez da função *WaitForSingleObject*.

Para compreender melhor o processamento que a função *WaitForSingleObject* executa, considere o programa *Wait_Events.cpp*, contido no CD-ROM que acompanha este livro. Toda vez que o usuário seleciona a opção de menu *Testar!*, o programa inicia um encadeamento que aguarda um evento, adormece e libera o evento. O uso de *eventos auto-reset* (eventos que mudam seus sinalizadores automaticamente) impõe o acesso do encadeamento em ordem serial porque os eventos reinicializam automaticamente seus valores para não-sinalizados quando o primeiro encadeamento em espera adquire o objeto. O programa *Wait_Events.cpp* efetua seu processamento operativo dentro das funções *ChildThreadProc* e *WndProc*.

USANDO WAITFORMULTIPLEOBJECT PARA SINCRONIZAR MUITOS ENCADEAMENTOS 1409

Você já aprendeu como seus programas podem usar a função *WaitForSingleObject* para sincronizar um encadeamento com um único evento de auto-reset. No entanto, mais comumente, seus programas irão requerer que o processamento continue apenas quando um ou mais de um certo conjunto de objetos ocorra. Nesses casos, seus

programas podem usar a função *WaitForMultipleObjects*. A função *WaitForMultipleObjects* retorna quando um dos seguintes eventos ocorre:

- Qualquer um ou todos os objetos especificados estão no estado sinalizado.
- O intervalo de tempo-limite transcorre.

Você usará a função *WaitForMultipleObjects* dentro de seus programas, como segue:

```
DWORD WaitForMultipleObjects(
   DWORD nConta,         // número de indicativos na matriz
                         // de indicativo de objetos
   CONST HANDLE *lpIndic, // ponteiro para a matriz de
                          // indicativos de objetos
   BOOL bAguardaTodos,    // sinalizador de espera
   DWORD dwMilisseg        // intervalo de tempo-limite em
                          // milissegundos
);
```

A função *WaitForMultipleObjects* aceita os parâmetros detalhados na Tabela 1409.1.

Tabela 1409.1 Os parâmetros para a função ***WaitForMultipleObjects***

Parâmetro	Descrição
nConta	Especifica o número de indicativos de objeto na matriz para a qual *lpIndic* aponta. O número máximo de indicativos de objetos é *MAXIMUM_WAIT_OBJECT* (uma constante definida pelo sistema que variará de uma instalação para outra).
lpIndic	Aponta para uma matriz de indicativos de objetos. A Tabela 1408.1 já listou os tipos de objeto cujos indicativos você pode especificar. A matriz *lpIndic* pode conter indicativos de objetos de diferentes tipos.
Nota: Sob o Windows NT, os indicativos precisam ter o direito de acesso SYNCHRONIZE.	
bAguardaTodos	Especifica o tipo de espera. Se for *True*, a função retornará quando o estado de todos os objetos na matriz *lpIndic* for sinalizado. Se for *False*, a função retornará quando o estado de qualquer um dos objetos estiver definido para sinalizado e o valor de retorno, então, indicará o objeto cujo estado fez a função retornar.
dwMilisseg	Especifica o intervalo de tempo-limite em milissegundos. A função retornará se o intervalo transcorrer, mesmo se as condições que o parâmetro *bAguardaTodos* especificou não forem encontradas. Se *dwMilisseg* for zero, a função testará os estados dos objetos especificados e retornará imediatamente. Se *dwMilisseg* for *INFINITE*, o intervalo de tempo-limite da função nunca transcorrerá.

Após ser chamada, *WaitForMultipleObjects* retornará no caso de falha, sucesso ou tempo-limite. Se a função falhar, o valor de retorno será *WAIT_FAILED*. Se a função for bem-sucedida, o valor de retorno indicará o evento que fez a função retornar. O valor de retorno, em caso de sucesso, é um dos valores detalhados na Tabela 1408.2.

A função *WaitForMultipleObjects* determina se um ou mais dos objetos que o encadeamento está aguardando atendeu ao critério de espera. Se nenhum dos objetos atendeu ao critério de espera, o encadeamento chamador entra em um estado de espera eficiente, consumindo muito pouco tempo do processador enquanto espera que um ou mais dos objetos que o encadeamento está esperando atenda ao critério de espera. Quando o parâmetro *bEsperaTudo* for *True*, a operação de espera da função será completada somente quando os estados de todos os objetos forem sinalizados. O parâmetro *bEsperaTodos* não modificará os estados dos objetos especificados até que os estados de todos os objetos estejam sinalizados. Por exemplo, um mutex pode ser sinalizado, mas o encadeamento não obtém a posse até que os estados dos outros objetos também estejam sinalizados. Enquanto isso, algum outro encadeamento pode obter a posse do mutex, desse modo definindo seu estado para não-sinalizado. Antes de retornar uma função de espera modifica o estado de alguns tipos de objetos de sincronização. A modificação ocorre somente para o objeto ou objetos cujo estado sinalizado fez a função retornar. Por exemplo, algum outro encadeamento ou processo decrementa o contador de um objeto semáforo em 1. A função

WaitForMultipleObjects pode especificar na matriz *lpIndic* um ou mais indicativos de quaisquer um dos tipos de objetos listados na Tabela 1408.1.

Exatamente como com a função *WaitForSingleObject*, você precisa ser cuidadoso ao usar a função *WaitForSingleObject* e o intercâmbio dinâmico de dados. Se um encadeamento criar quaisquer janelas, ele precisará processar mensagens. O intercâmbio dinâmico de dados envia mensagens para todas as janelas no sistema. Se você tiver um encadeamento que usa uma função de espera sem intervalo de tempo-limite, o sistema entrará em um impasse. Portanto, se você tiver um encadeamento que cria janelas, use as funções *MsgWaitForMultipleObjects* ou *MsgWaitForMultipleObjectsEx*, em vez da função *WaitForMultipleObjects*.

CRIANDO UM MUTEX 1410

Como você aprendeu, os mutexes são similares às seções críticas, exceto que seus programas também podem usar mutexes para sincronizar acessos de dados entre múltiplos objetos, em vez de dentro somente de um único objeto. Para usar um mutex, seus programas precisam primeiro criar o mutex com a função *CreateMutex*. Você usará a função *CreateMutex* dentro de seus programas, como mostra o protótipo a seguir:

```
HANDLE CreateMutex(
   LPSECURITY_ATTRIBUTES lpAtribMutex,  // atributos de segurança
   BOOL bProprietInicial, // sinalizador para posse inicial
   LPCTSTR  lpNome        // ponteiro para nome do objeto mutex
);
```

O parâmetro *lpAtribMutex* é um ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho poderão herdar o indicativo retornado. Se *lpAtribMutex* for *NULL*, os processos-filho não poderão herdar o indicativo. Sob o Windows NT, o membro *lpSecurityDescriptor* da estrutura especifica um descritor de segurança para o novo mutex. Se *lpAtribMutex* for *NULL*, o mutex recebe um descritor de segurança padrão. Sob o Windows 95, a função *CreateMutex* ignora o membro *lpSecurityDescriptor* da estrutura. O parâmetro *bProprietInicial* especifica o proprietário inicial do objeto mutex. Se for *True*, o encadeamento chamador irá requerer a posse imediata do objeto mutex. Caso contrário, nenhum mutex possui o objeto mutex. O parâmetro *lpNome* aponta para uma string terminada por *NULL* que especifica o nome do objeto mutex. O Windows limita o nome a *MAX_PATH* caracteres, e o nome pode conter qualquer caractere exceto a barra invertida (\). Lembre-se de que a comparação de nome não distingue o tipo de letra.

Se o parâmetro *lpNome* for igual ao nome de um objeto mutex existente, *lpNome* solicitará *MUTEX_ALL_ACCESS* ao objeto existente. A função *lpNome* ignora o parâmetro *bProprietInicial*, pois o processo de criação já o definiu. Se o parâmetro *lpMutexAtrib* não for *NULL*, *CreateMutex* determinará se o indicativo é herdável, mas ignorará o membro de descritor de segurança do indicativo. Se *lpNome* for *NULL*, *CreateMutex* criará o objeto mutex sem um nome. Se a função *lpNome* for igual ao nome de um objeto *evento*, *semáforo* ou mapeado em arquivo existente, ela falhará e a função *GetLastError* retornará o acesso *ERROR_INVALID_HANDLE*. A função falha porque os objetos evento, mutex, semáforo e mapeado em arquivo compartilham o mesmo *namespace*.

O indicativo que *CreateMutex* retorna tem acesso *MUTEX_ALL_ACCESS* para o novo objeto mutex, e você pode usá-lo em qualquer função que requeira um indicativo para um objeto mutex. Qualquer encadeamento de processo chamador pode especificar o indicativo de objeto mutex em uma chamada a uma das funções de espera (tais como *WaitForSingleObject* e *WaitForMultipleObjects*). As funções de espera de objeto único retornarão quando o estado do objeto especificado for sinalizado. Você pode instruir as funções de espera de múltiplos objetos a retornar quando qualquer um ou quando todos os objetos especificados forem sinalizados. Quando uma função de espera retorna, o sistema operacional libera o encadeamento em espera para continuar sua execução.

O estado de um objeto mutex está sinalizado quando nenhum encadeamento o possui. O encadeamento criador pode usar o sinalizador *bProprietInicial* para solicitar posse imediata do mutex. Caso contrário, um encadeamento precisa usar uma das funções de espera para solicitar a posse. Quando o estado do mutex for sinalizado, o sistema operacional concede a um encadeamento em espera posse, o estado do mutex muda para não-sinalizado e a função de espera retorna. Somente um encadeamento pode possuir um mutex em um determinado momento. O encadeamento possuidor usa a função *ReleaseMutex* para abrir mão da posse. O encadeamento que possui um mutex pode especificar o mesmo mutex em chamadas de funções de espera repetidas sem bloquear sua execução. Tipicamente, um encadeamento não esperará repetidamente o mesmo mutex, mas o me-

canismo de chamada repetida impedirá um encadeamento de criar um impasse consigo mesmo ao aguardar um mutex que ele já possui. No entanto, para liberar sua posse, o encadeamento precisa chamar a função *ReleaseMutex* uma vez sempre que o mutex satisfez uma espera.

Dois ou mais processos podem chamar *CreateMutex* para criar o mesmo objeto mutex nomeado. O primeiro processo cria o mutex, e os processos subsequentes abrem um indicativo para o mutex existente. Múltiplos processos podem usar *CreateMutex* para lhes permitir obter indicativos do mesmo objeto mutex, ao mesmo tempo aliviando o usuário da responsabilidade de garantir que o processo criador iniciou o mutex anteriormente. Ao usar *CreateMutex* em múltiplos processos, você deve definir o sinalizador *bPropInicial* como *False*; caso contrário, pode ser difícil saber qual processo tem a posse inicial. Múltiplos processos podem ter indicativos do mesmo objeto mutex, permitindo que seus programas usem o objeto mutex para a sincronização interprocesso. Os seguintes mecanismos de compartilhamento de objeto estão disponíveis para os seus programas:

- O processo-filho que a função *CreateProcess* cria pode herdar um indicativo para um objeto mutex se o parâmetro *lpMutexAtributos* de *CreateMutex* habilitou a herança.
- Um processo pode especificar o indicativo *objeto mutex* em uma chamada à função *DuplicateHandle* para criar um indicativo duplicado que outro processo possa usar.
- Um processo pode especificar o nome de um objeto mutex em uma chamada às funções *OpenMutex* ou *CreateMutex*.

Use a função *CloseHandle* para fechar o indicativo. O sistema fecha o indicativo automaticamente quando o processo termina. O sistema operacional destrói o objeto mutex quando o sistema fecha seu último indicativo para o mutex.

1411 USANDO UM MUTEX DENTRO DE UM PROGRAMA DE EXEMPLO

Seus programas podem criar facilmente objetos mutex para ajudar com a sincronização de encadeamentos entre múltiplos processos. Na dica anterior, você aprendeu o processo básico subjacente à criação de um objeto mutex. No entanto, após você criar um mutex, seus programas precisam então obter um indicativo para esse mutex antes de poderem usá-lo dentro de seu próprio código. Como você aprendeu, um segundo processo pode obter um indicativo a um mutex chamando um comando *CreateMutex* com o mesmo mutex nomeado que o primeiro processo. Alternativamente, seus programas podem usar a função *OpenMutex* para obter um indicativo para um objeto mutex criado anteriormente. Você usará a função *OpenMutex* dentro de seus programas, como mostrado no protótipo a seguir:

```
HANDLE OpenMutex(
   DWORD dwAcessoDesejado,  // sinalizador de acesso
   BOOL  hIndicHeranca,     // sinalizador de herança
   LPCTSTR lpNome           // ponteiro para nome objeto mutex
);
```

O parâmetro *dwAcessoDesejado* especifica o acesso solicitado para o objeto mutex. Para os sistemas que suportam objetos de segurança (tal como as instalações seguras do Windows NT), o parâmetro falha se o descritor de segurança do objeto especificado não permite o acesso solicitado para o processo chamador. O parâmetro *dwAcessoDesejado* pode ser qualquer combinação de valor *MUTEX_ALL_ACCESS*, que especifica todos os sinalizadores de acesso possíveis para o objeto mutex; o valor *SYNCHRONIZE*, que somente o Windows NT suporta e que permite que um processo use o indicativo de mutex em qualquer uma das funções de espera para adquirir a posse do objeto mutex, ou na função *ReleaseMutex* para liberar a posse; ou ambos.

A função *OpenMutex* permite que múltiplos processos abram indicativos do mesmo objeto mutex. A função *OpenMutex* é bem-sucedida somente se algum processo já tiver criado o mutex, usando a função *CreateMutex*. O processo chamador pode usar o indicativo retornado em quaisquer funções que requeiram um indicativo de objeto mutex, tal como as funções de espera, que estão sujeitas às limitações do acesso que o parâmetro *dwAcessoDesejado* especifica.

Seus programas podem usar a função *DuplicateHandle* para duplicar um indicativo, e a função *CloseHandle* para fechar o indicativo. O sistema fecha o indicativo automaticamente quando o processo termina. O sistema operacional destrói o objeto mutex quando o sistema fecha o último indicativo do mutex.

Para compreender melhor o processamento que seus programas efetuarão com os objetos mutex, considere o programa *Simple_Mutex.cpp* no CD-ROM que acompanha este livro. O programa *Simple_Mutex.cpp* cria um objeto mutex simples. Quando o usuário seleciona o item de menu *Testar!*, o programa inicia um encadeamento que aguarda acessos para um mutex, adormece e depois libera o mutex. Se o usuário selecionar a opção *Testar!* múltiplas vezes, o programa demonstrará *mutex contention*. Isto é, o mutex forçará os encadeamentos subsequentes a esperar até que cada encadeamento precedente complete sua execução. Embora o programa *Simple_Mutex.cpp* crie o encadeamento-filho dentro da função *WndProc*, a maioria do processamento operativo do programa ocorre dentro da função *ChildThreadProc*.

USANDO SEMÁFOROS 1412

Como você aprendeu, a maioria dos semáforos usa um contador para a sincronização. Dentro de seus programas, você geralmente usará semáforos para limitar o acesso a um objeto, algum código ou outro recurso limitado. Quando você criar um semáforo, dirá ao semáforo quantos acessos ele deverá permitir e qual seu número inicial de acessos. Você usará a função *CreateSemaphore* para criar um semáforo. O protótipo para ela é o seguinte:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpAtribSemaf, // atributos de segurança
  LONG lContInicial,    // contador inicial
  LONG lContMax,        // contador máximo
  LPCTSTR  lpNome       // ponteiro para nome de semáforo
);
```

A função *CreateSemaphore* aceita os parâmetros detalhados na Tabela 1412.

Tabela 1412 Parâmetros para a função ***CreateSemaphore***.

Parâmetro	Descrição
lpAtribSemaf	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho poderão herdar o indicativo retornado. Se *lpAtribSemaf* for *NULL*, os processos-filho não poderão herdar o indicativo. Sob o Windows NT, o membro *lpSecurityDescriptor* da estrutura especifica um descritor de segurança para o novo semáforo. Se *lpAtribSemaf* for *NULL*, o semáforo receberá um descritor de segurança padrão. Sob o Windows 95, a função ignora o membro *lpSecurityDescriptor* da estrutura.
lContInicial	Especifica um contador inicial para o objeto semáforo. Esse valor precisa ser maior ou igual a zero e menor ou igual a *lContMax*. O estado de um semáforo será sinalizado quando seu contador for maior que zero, e não sinalizado quando ele for zero. A função de espera decrementa o contador sempre que libera um encadeamento que estava esperando o semáforo. A função de espera incrementa o contador por uma quantia especificada chamando a função *ReleaseSemaphore*
lContMax	Especifica o contador máximo para o objeto semáforo. Esse parâmetro precisa ser maior que zero
lpNome	Aponta para uma string terminada por *NULL* que especifica o nome do objeto semáforo. O Windows limita o nome a *MAX_PATH* caracteres, e o nome pode conter qualquer caractere exceto a barra invertida (\) separadora. Lembre-se de que a comparação de nome faz distinção na caixa das letras.

Tabela 1412 Parâmetros para a função ***CreateSemaphore*** (*Continuação*)

Parâmetro	Descrição
	Se lpNome for igual ao nome de um objeto semáforo nomeado existente, *lpNome* solicitará *SEMAPHORE_ALL_ACCESS* para o objeto existente. A função ignora os parâmetros *lContInicial* e *lContMax*, porque o processo criador já os definiu. Se o parâmetro *lpAtribSemaf* não for *NULL*, ele determinará se o indicativo será herdável, mas a função ignorará o membro do descritor de segurança do parâmetro. Se *lpNome* for *NULL*, a função criará o objeto semáforo sem um nome. Se *lpNome* for igual ao nome de um evento existente, mutex ou objeto de mapeamento de arquivo, a função falhará e a função *GetLastError* retornará a constante *ERROR_INVALID_HANDLE*. Isso ocorre porque os objetos evento, mutex, semáforo e de mapeamento de arquivo compartilham o mesmo *namespace*.

Se a função *CreateSemaphore* for bem-sucedida, o valor de retorno será um indicativo para o objeto semáforo. Se o objeto semáforo existia antes da chamada da função, *GetLastError* retorna a constante *ERROR_ALREADY_EXISTS*. Se a função falhar, ela retornará *NULL*.

O indicativo que a função *CreateSemaphore* retorna tem acesso *SEMAPHORE_ALL_ACCESS* para o novo objeto semáforo, e seus programas podem usá-la em qualquer função que requeira um indicativo para um objeto semáforo. Qualquer encadeamento de processo chamador pode especificar o indicativo objeto semáforo em uma chamada a uma das funções de espera. As funções de espera de um único objeto retornarão quando o estado do objeto especificado estiver sinalizado. Seus programas podem instruir as funções de espera de múltiplos objetos a retornarem quando qualquer uma ou quando todos os objetos sinalizados estiverem sinalizados. Quando uma função de espera retorna, o sistema operacional libera o encadeamento de espera para continuar sua execução.

O estado de um objeto semáforo será sinalizado quando seu contador for maior que zero, e não-sinalizado quando seu contador for igual a zero. O parâmetro *lContaInicial* especifica o contador inicial. Toda vez que o sistema libera um encadeamento em espera por causa do estado sinalizado do semáforo, o semáforo decrementa seu contador em 1. Use a função *ReleaseSemaphore* para incrementar o contador de um semáforo por uma quantia especificada. O contador nunca pode ser menor que zero ou maior que o valor especificado no parâmetro *lContaMax*.

Múltiplos processos podem ter indicativos do mesmo objeto semáforo, deixando que os processos usem o objeto para a sincronização interprocesso. Os seguintes mecanismos de compartilhamento de objetos estão disponíveis para os seus programas:

- Um processo-filho que a função *CreateProcess* criou pode herdar um indicativo para um objeto semáforo se o parâmetro *lpAtribSemaf* de *CreateSemaphore* habilitou a herança.
- Um processo pode especificar o indicativo *objeto semáforo* em uma chamada à função *DuplicateHandle* para criar um indicativo duplicado que outro processo pode usar.
- Um processo pode especificar o nome de um objeto semáforo em uma chamada às funções *OpenSemaphore* ou *CreateSemaphore*.

Use a função *CloseHandle* para fechar o indicativo. O sistema fecha o indicativo automaticamente quando o processo termina. O sistema operacional destrói um objeto semáforo quando o sistema fecha o último indicativo para o semáforo.

Como você aprendeu na dica anterior, seus programas usam um mutex criando, abrindo e liberando o mutex. Você executará passos similares com um semáforo. Para compreender melhor o processamento que seus programas podem efetuar com um semáforo, considere o programa *Create_Semaphore.cpp* no CD-ROM que acompanha este livro. Esse programa usa semáforos para garantir que somente quatro encadeamentos de cada vez podem executar o procedimento do encadeamento-filho. Toda vez que o usuário seleciona a opção de menu *Testar!*, o programa tenta criar um novo encadeamento. No entanto, o procedimento do encadeamento garante que existem pontos de encadeamentos (até quatro) antes de permitir que o encadeamento esteja executando. Se não existirem recursos disponíveis para o semáforo, o procedimento de encadeamento aguardará até que um processo se torne disponível. O processamento operativo ocorre dentro da função *ChildThreadProc* do programa *Create_Semaphore.cpp*.

USANDO UM PROCESSADOR DE EVENTO SIMPLES 1413

Como você aprendeu, eventos são a forma mais primitiva de objetos de sincronização. Seus programas geralmente usarão um evento para sinalizar um ou mais encadeamentos que uma operação completou. Exatamente como os semáforos e os mutexes, seus programas criarão um evento chamando a função *CreateEvent*. Você usará a função *CreateEvent* dentro de seus programas como segue:

```
HANDLE CreateEvent(
   LPSECURITY_ATTRIBUTES lpAtribEventos,  // atributos de segurança
   BOOL bResetManual,  // sinalizador para o evento reinicialização manual
   BOOL bEstadoInic,   // sinalizador para estado inicial
   LPCTSTR lpNome      // ponteiro para nome de objeto evento
```

A função *CreateEvent* aceita os parâmetros detalhados na Tabela 1413.

Tabela 1413 Os parâmetros para a função ***CreateEvent***.

Parâmetro	Descrição
lpAtribEventos	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribEvento* for *NULL*, o indicativo não será herdável. Sob o Windows NT, o membro *lpSecurityDescriptor* da estrutura especifica um descritor de segurança para o novo evento. Se *lpAtribEvento* for *NULL*, o evento obterá um descritor de segurança padrão. Sob o Windows 95, *CreateEvent* ignora o membro *lpSecurityDescriptor* da estrutura.
bResetManual	Especifica se *CreateEvent* cria um objeto evento de reinicialização manual ou automática. Se for *True*, você precisará usar a função *ResetEvent* para reinicializar manualmente o estado para não-sinalizado. Se for *False*, o Windows automaticamente reinicializará o estado para não-sinalizado após o sistema ter liberado um único encadeamento de espera.
bEstadoInicial	Especifica o estado inicial do objeto evento. Se for *True*, o estado inicial será inicializado; caso contrário, será não-sinalizado.
lpNome	Aponta para uma string terminada por *NULL* que especifica o nome do objeto evento. O Windows limita o nome para *MAX_PATH* caracteres, e o nome pode conter qualquer caractere exceto a barra invertida (\). Lembre-se de que a comparação de nome distingue a caixa das letras. Se *lpNome* for igual ao nome de um objeto evento existente nomeado, *lpNome* solicitará *EVENT_ALL_ACCESS* ao objeto existente *CreateEvent* ignora os parâmetros *bResetManual* e *bEstadoInicial* porque o processo criador já os definiu. Se o parâmetro *lpAtribEvento* não for *NULL*, ele determinará se o indicativo será herdável, mas *CreateEvent* ignorará seu membro descritor de segurança. Se *lpNome* for *NULL*, *CreateEvent* criará o objeto evento sem um nome. Se *lpNome* for igual ao nome de um semáforo, mutex ou objeto de mapeamento de arquivo existente, a função falhará e *GetLastError* retornará a constante *ERROR_INVALID_HANDLE*. O erro ocorre porque os objetos evento, mutex, semáforo e de mapeamento de arquivo compartilham o mesmo *namespace*.

Se a função *CreateEvent* for bem-sucedida, o valor de retorno será um indicativo para o objeto evento. Se o objeto evento nomeado existia antes da chamada da função, a função *GetLastError* retorna a constante *ERROR_ALREADY_EXISTS*. Se a função falhar, o valor de retorno será NULL.

O indicativo que *CreateEvent* retorna tem o direito de acesso *EVENT_ALL_ACCESS* ao novo objeto evento, e qualquer função que requeira um indicativo para um objeto evento poderá usá-lo. Qualquer encadeamento de processo chamador pode especificar o indicativo do objeto evento em uma chamada a uma das funções de espera. As funções de espera de um único objeto retornam quando o estado do objeto especificado está sinalizado. Você pode instruir as funções de espera de múltiplos objetos a retornar quando qualquer uma ou quando

todos os objetos especificados são sinalizados. Quando uma função de espera retorna, o sistema operacional libera o encadeamento de espera para continuar sua execução.

O parâmetro *bEstadoInicial* especifica o estado inicial do objeto evento. Use a função *SetEvent* para definir o estado de um objeto evento como sinalizado. Use a função *ResetEvent* para reinicializar o estado de um objeto evento para não-sinalizado. Quando o estado de um objeto evento de reinicialização manual está sinalizado, ele permanece sinalizado até que a função *ResetEvent* explicitamente o reinicie para não-sinalizado. Seus programas poderão liberar qualquer número de encadeamentos de espera, ou encadeamentos que subseqüentemente iniciarão as operações de espera para o objeto evento especificado, enquanto o estado do objeto for sinalizado.

Quando o estado de um objeto evento de auto-reinicialização for sinalizado, ele permanecerá sinalizado até que seus programas, ou o sistema operacional, liberarem um único encadeamento de espera; o sistema, então, automaticamente, reiniciará o estado para não-sinalizado. Se nenhum encadeamento estiver aguardando, o estado do objeto evento permanecerá sinalizado. Múltiplos processos podem ter indicativos do mesmo objeto evento, habilitando o uso do objeto para a sincronização interprocesso. Os mecanismos de compartilhamento de objetos a seguir estão disponíveis para os seus programas:

- Um processo-filho que a função *CreateProcess* criou pode herdar um indicativo para um objeto evento se o parâmetro *lpAtribEvento* de *CreateEvent* habilitou a herança.
- Um processo pode especificar o indicativo do objeto evento em uma chamada à função *DuplicateHandle* para criar um indicativo duplicado que outro processo pode usar.
- Um processo pode especificar o nome do objeto evento em uma chamada às funções *OpenEvent* ou *CreateEvent*.

Use a função *CloseHandle* para fechar o indicativo. O sistema fecha o indicativo imediatamente quando o processo termina. O sistema operacional destrói o objeto evento quando o sistema fecha seu último indicativo.

Para compreender melhor o processamento que seus programas executam quando trabalham com eventos, considere o programa *Three_Options.cpp*, contido no CD-ROM que acompanha este livro. Para ver melhor os efeitos da sincronização, rode o programa *Three_Options* três vezes e disponha as ocorrências lado a lado na área de trabalho. Nas duas primeiras ocorrências, selecione a opção de menu Read, e, na terceira ocorrência, selecione a opção de menu Write. Embora os leitores possam executar simultaneamente, o escritor precisa aguardar até que os leitores terminem. Após as operações de leitura terminarem, a operação de escrita é executada. Se você inverter a ordem e selecionar a operação de escrita primeiro, as operações de leitura precisarão aguardar até que a operação de escrita complete seu processamento. Cada processo mostra seu estado atual dentro da barra de status da janela. O programa *Three_Options.cpp* completa o processamento operativo dentro da função *WndProc*.

1414 COMPREENDENDO A INTERFACE DE DISPOSITIVO GRÁFICO

A interface de dispositivo gráfico (GDI) é um conjunto de funções de biblioteca que fornece aplicativos Windows com uma interface independente do dispositivo para a tela e para as impressoras. A interface de dispositivo gráfico é um nível entre um aplicativo e os diferentes tipos de hardware. A interface de dispositivo gráfico libera o programa de ter que lidar com cada tipo de dispositivo diretamente, permitindo que a interface de dispositivo gráfico resolva as diferenças no hardware. Um aplicativo Windows bem projetado funcionará de forma igual em todos os tipos de hardware atuais e em todos os novos que os fabricantes lançarem no mercado devido a sua interface de dispositivo gráfico.

Todas as funções da interface de dispositivo gráfico em Win32 usam valores de 32 bits para as coordenadas de interface de dispositivo gráfico; no entanto, no Windows 95 e no Win32s, o sistema operacional ignora a palavra de alta ordem, o que resulta em um valor de 16 bits para as coordenadas. Somente no Windows NT seus aplicativos usam todos os valores de 32 bits.

1415 RAZÕES PARA USAR A INTERFACE DE DISPOSITIVO GRÁFICO

Como você pode imaginar, seus programas devem usar a interface de dispositivo gráfico para gerar saída em vez de usar saída textual ou não-gráfica. Existem muitas razões para usar a interface de dispositivo gráfico para manipular o conteúdo de uma janela, incluindo as seguintes:

- Você pode aplicar o mesmo dispositivo de contexto para múltiplos dispositivos.
- Você pode formatar saída no dispositivo do contexto antes de enviá-la para o dispositivo.
- Você pode manipular gráficos e outras informações visuais dentro da janela com um dispositivo de contexto.
- Você pode controlar a aparência da janela (após rolar, ajustar o tamanho e assim por diante) mais facilmente com um dispositivo do contexto.
- Seus programas podem facilmente enviar a saída que você colocou anteriormente em um dispositivo de contexto de janela ou em uma impressora a outro dispositivo de contexto.

Muitos dos programas que você usou anteriormente neste livro usaram os dispositivos de contexto para manter informações dentro da janela do programa. Nas dicas a seguir, enfocaremos mais os méritos e os usos dos dispositivos de contexto.

COMPREENDENDO MELHOR OS DISPOSITIVOS DO CONTEXTO 1416

A ferramenta básica que o Windows usa para fornecer independência do dispositivo para um aplicativo é um *dispositivo do contexto*, ou DC. O dispositivo do contexto é uma estrutura interna que o Windows usa para manter informações sobre um dispositivo do contexto. Em vez de enviar saída diretamente para o hardware, o aplicativo envia a saída para o dispositivo do contexto. O Windows, então, a envia para o hardware.

Um dispositivo do contexto sempre contém uma caneta para desenhar linhas, um pincel para preencher áreas, uma fonte para saída de caracteres e uma série de outros valores para controlar como o dispositivo do contexto se comporta. Se o aplicativo requer uma fonte diferente, o aplicativo deve selecionar a fonte em um dispositivo do contexto antes de exibir o texto. Selecionar uma nova fonte não altera o texto existente na área cliente em uma janela.

Você pode visualizar a interface que um dispositivo do contexto fornece como sendo similar ao mostrado na Figura 1416.

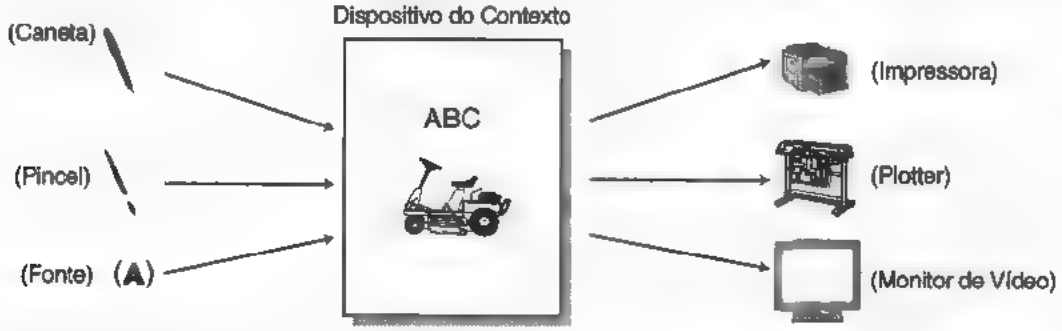

Figura 1416 Modelo lógico para um dispositivo do contexto.

USANDO DISPOSITIVOS DO CONTEXTO PRIVADOS 1417

Normalmente, os aplicativos compartilham e recuperam os dispositivos do contexto antes de usar, e os liberam após o uso. Tratar os dispositivos do contexto por somente períodos curtos de tempo funciona melhor para aplicativos que não usam freqüentemente um dispositivo do contexto. Um aplicativo que requer um dispositivo do contexto pode criar uma janela com seu próprio dispositivo do contexto privado, especificando o estilo de classe *CS_OWNDC* na definição de classe para a janela. Com o estilo de classe *CS_OWNDC*, o dispositivo do contexto existe durante toda a vida de uma janela. Um aplicativo ainda usará a função *GetDC* para recuperar um indicativo para o dispositivo do contexto, mas não requer uma chamada à função *ReleaseDC* após ela completar seu processamento do dispositivo do contexto. Quando você usa um dispositivo do contexto privado com programas que alteram as definições do dispositivo do contexto, pode criar modificações, tais como novas cores de texto, de canetas e de pincéis, que permanecem em efeito até que o programa as altere novamente.

1418 COMPREENDENDO AS ORIGENS E AS EXTENSÕES

Um dispositivo do contexto tem dois modos especiais de mapeamento, *MM_ISOTROPIC* e *MM_ANISOTROPIC*, que um tamanho fixo não restringe. Esses modos de mapeamento usam duas regioes retangulares, a *janela* e o *viewport*, para derivar um fator de escala e uma orientação. A janela está em coordenadas lógicas, e o viewport está em coordenadas físicas. Juntos, eles determinam como o sistema operacional mapeia unidades lógicas em unidades físicas. Tanto a janela quanto o viewport contêm uma *origem*, uma *extensão X* e uma *extensão Y*. A *origem* é um ponto que descreve qualquer um dos quatro vértices. A origem do viewport é um deslocamento a partir da *origem da janela*. A *extensão X* é a distância horizontal da origem até seu vértice oposto. A *extensão Y* é a distância vertical da origem até seu vértice oposto.

O Windows define um fator de escala horizontal dividindo a extensão X do viewport pela extensão X da janela O Windows define um fator de escala vertical dividindo a extensão Y do viewport pela extensão Y da janela. Esses fatores de escala determinam o número de unidades lógicas que o Windows mapeia para um número de pixels. Além de determinar os fatores de escala, a janela e o viewport determinam a orientação de um objeto.

1419 OBTENDO O DISPOSITIVO DE CONTEXTO PARA UMA JANELA

Como você aprendeu, seus programas devem usar dispositivos de contexto para criar telas gráficas e de texto, tanto em um dispositivo janela quanto em um dispositivo impressora. Seus programas podem usar a função *GetDC* ou *GetDCEx* para recuperar um dispositivo de contexto para uma janela. A função *GetDCEx* recupera o indicativo de um dispositivo de contexto de tela nas funções de interface de dispositivo gráfico para desenhar na área cliente. Seus programas podem usar o dispositivo de contexto de tela em funções de interface de dispositivos gráficos para desenhar na área cliente. A função *GetDCEx* é uma extensão para a função *GetDC* que dá a um aplicativo maior controle sobre se e como o corte ocorre na área cliente. Dentro de seus programas, você usará a função *GetDCEx*, como segue:

```
HDC GetDCEx(
   HWND hJan,        // indicativo da janela
   HRGN hrgnCorte,   // indicativo da região de corte
   DWORD sinaliz     // sinalizadores de criação de dispositivo de contexto
);
```

O parâmetro *hJan* identifica a janela onde o desenho ocorrerá. O parâmetro *hrgnCorte* especifica uma região de corte que você pode combinar com a região visível da janela cliente. O parâmetro *sinaliz* especifica uma região de corte que você pode combinar com a região visível da janela cliente. Os parâmetros *sinaliz* especificam como um aplicativo criará um dispositivo de contexto. Os parâmetros *sinaliz* podem ser uma combinação dos valores listados pela Tabela 1419.

Tabela 1419 Os valores possíveis para o parâmetro *sinaliz*.

Valor	Significado
DCX_WINDOW	Retorna um dispositivo do contexto que corresponde ao retângulo da janela em vez de ao retângulo cliente.
DCX_CACHE	Retorna um dispositivo do contexto do cache, em vez da janela *OWNDC* ou *CLASSDC*. Essencialmente, esse valor anula *CS_OWNDC* e *CS_CLASSDC*.
DCX_PARENTECLIP	Usa a região visível da janela-mãe. Esse valor ignora os bits de estilo *WS_CLIPCHILDREN* e *CS_PARENTDC* da mãe. *GetDCEx* define a origem do dispositivo do contexto no canto superior esquerdo da janela que *hJan* identifica.
DCX_CLIPSIBLINGS	Exclui as regiões visíveis de todas as janelas-irmã acima da janela que *hJan* identifica.
DCX CLIPCHILDREN	Exclui as regiões visíveis de todas as janelas-filha abaixo da janela que *hJan* identifica.

Tabela 1419 Os valores possíveis para o parâmetro ***sinaliz*** *(Continuação)*

Valor	Significado
DCX_NORESETATTRS	Não reinicializa os atributos desse dispositivo do contexto para os atributos padrão quando o programa libera esse dispositivo do contexto.
DCX_LOCKWINDOWUPDATE	Permite que o programa e o usuário desenhem no dispositivo do contexto mesmo se houver uma chamada *LockWindowUpdate* em efeito, o que, em caso contrário, excluiria essa janela. Você usará esse valor para permitir que seus programas desenhem durante uma operação de *tracking*
DCX_EXCLUDERGN	Exclui a região de corte que *hrgnCorte* identifica a partir da região visível do dispositivo do contexto que *GetDCEx* retorna.
DCX_INTERSECTRGN	Intercepta a região de corte que *hrgnCorte* identifica com a região visível do dispositivo do contexto que *GetDCEx* retorna.
DCX_VALIDATE	Quando especificado com *DCX_INTERSECTUPDATE*, faz o dispositivo do contexto que *GetDCEx* retorna ser completamente validado. Usar essa função com *DCX_INTERSECTUPDATE* e *DCX_VALIDATE* é idêntico a usar a função *BeginPaint*.

A menos que o dispositivo do contexto de exibição pertença a uma classe de janela, o aplicativo precisa chamar a função *ReleaseDC* para liberar o dispositivo do contexto após a pintura. Como somente cinco dispositivos do contexto comuns estão disponíveis para seus programas a qualquer tempo, deixar de liberar um dispositivo do contexto pode impedir que outros programas o acessem. Tanto as funções *GetDC* quanto *GetDCEx* retornam um dispositivo do contexto que pertence à classe de janela se o programa especificou *CS_CLASSDC*, *CS_OWNDC* ou *CS_PARENTDC* como um estilo na estrutura *WNDCLASS* quando ele registrou a classe.

Para compreender melhor o processamento que a função *GetDCEx* executa, considere o programa *Draw_Hollow.cpp*, contido no CD-ROM que acompanha este livro. O programa usa a função *GetDCEx* para obter um dispositivo do contexto para a área cliente da janela, excluindo uma região. O programa então pinta um retângulo cinza na área cliente, excluindo a região interna (branca). O programa *Draw_Hollow.cpp* efetua o processamento operativo dentro da função *WndProc*

CRIANDO UM DISPOSITIVO DO CONTEXTO PARA UMA IMPRESSORA 1420

Como você aprendeu, seus programas devem usar dispositivos do contexto para desenhar em um dispositivo. Embora as janelas tenham um dispositivo do contexto interno (ou privado ou público), outros dispositivos de saída não têm conteúdo preexistente. Seus programas devem usar a função *CreateDC* para criar um dispositivo do contexto para um dispositivo usando o nome que você especificar. Você usará a função *CreateDC* dentro de seus programas, como mostrado no protótipo a seguir:

```
HDC CreateDC(
  LPCTSTR  lpszContro, // ponteiro para uma string que
                       // especifica o nome do controlador
  LPCTSTR lpszDispo,   // ponteiro para uma string que
                       // especifica o nome do dispositivo
  LPCTSTR lpszSaida,   // não use; deixe  como NULL
  CONST DEVMODE *lpDadosInic   // ponteiro para dados opcionais da impressora
);
```

A função *CreateDC* aceita os parâmetros detalhados na Tabela 1420.1

Tabela 1420.1 Parâmetros para a função *CreateDC*.

Parâmetro	Descrição
lpszContro	Os aplicativos escritos para as versões anteriores do Windows usavam esse parâmetro para especificar o nome de arquivo (sem extensão) do controlador de dispositivo. Nos aplicativos baseados no Windows 95 e no Win32, CreateDC ignora esse parâmetro, que deve ser *NULL*, com uma exceção: você pode obter um dispositivo do contexto especificando a string terminada em *NULL DISPLAY*. Se esse parâmetro for *DISPLAY*, todos os outros parâmetros precisarão ser *NULL*. No ambiente Windows NT, *lpszContro* aponta para uma string de caracteres terminada por *NULL* que especifica *DISPLAY* para um controlador de vídeo, ou o nome de um controlador de impressão, que normalmente é *WINSPOOL*.
lpszDispo	Aponta para uma string de caracteres terminada por *NULL* que especifica o nome do dispositivo de saída específico que seus programas usarão. O Gerenciador de Impressão mostrará o nome do dispositivo de saída, tal como "Epson FX-80", que é um nome interno do Windows, não necessariamente o nome do modelo da impressora. Você precisa usar o parâmetro *lpszDispo*.
lpszSaida	O Windows ignora este parâmetro. Não o use em um aplicativo Win32. Os aplicativos baseados no Win32 devem definir esse parâmetro para *NULL*, pois pszSaida existe para oferecer compatibilidade para os aplicativos escritos para as versões anteriores do Windows.
lpInicDados	Aponta para uma estrutura *DEVMODE* que contém dados de inicialização específicos do dispositivo para o controlador de dispositivo. A função *DocumentProperties* lê essa estrutura e a preenche conforme for apropriado para o dispositivo especificado. O parâmetro *lpInicDados* precisará ser *NULL* se o controlador de dispositivo usar a *inicialização padrão que o usuário especificar*.

Se a função *CreateDC* for bem-sucedida, o valor de retorno será o indicativo para um dispositivo do contexto para o dispositivo especificado. Se *CreateDC* falhar, o valor de retorno será *NULL*.

Como você viu na Tabela 1420.1, a função *CreateDC* espera que seu último parâmetro seja um ponteiro para uma estrutura *DEVMODE* que contém informações de inicialização específicas do dispositivo para o controlador de dispositivo. A API Win32 define a estrutura *DEVMODE*, como mostrado aqui:

```
typedef struct _devicemode {
    TCHAR  dmDeviceName[32];
    WORD   dmSpecVersion;
    WORD   dmDriverVersion;
    WORD   dmSize;
    WORD   dmDriverExtra;
    DWORD  dmFields;
    short  dmOrientation;
    short  dmPaperSize;
    short  dmPaperLength;
    short  dmPaperWidth;
    short  dmScale;
    short  dmCopies;
    short  dmDefaultSource;
    short  dmPrintQuality;
    short  dmColor;
    short  dmDuplex;
    short  dmYResolution;
    short  dmTTOption;
    short  dmCollate;
    TCHAR  dmFormName[32];
    WORD   dmUnusedPadding;
    USHORT dmBitsPerPel;
    DWORD  dmPelsWidth;
    DWORD  dmPelsHeight;
```

```
    DWORD  dmDisplayFlags;
    DWORD  dmDisplayFrequency;
} DEVMODE;
```

A estrutura de dados *DEVMODE* contém informações sobre a inicialização do dispositivo e o ambiente de uma impressora. A Tabela 1420.2 detalha os membros da estrutura de dados *DEVMODE*.

Tabela 1420.2 Os membros da estrutura ***DEVMODE***.

Membros	Descrição
dmDeviceName	Especifica o nome do dispositivo que o controlador suporta (por exemplo, "PCL/HP LaserJet" no caso da PCL/HP *LaserJet*®). Essa string é única entre os controladores de dispositivo
dmSpecVersion	Especifica o número da versão da especificação de inicialização de dados na qual a estrutura do dispositivo do contexto está baseada.
dmDriverVersion	Especifica o número de versão do controlador de impressora que o desenvolvedor do controlador atribuiu.
dmSize	Especifica o tamanho, em bytes, da estrutura *DEVMODE*, exceto o membro *dmDriverData* (específico do dispositivo). Se um aplicativo manipula somente a porção independente do controlador dos dados, o aplicativo pode usar esse membro para determinar o comprimento de uma estrutura sem ter que levar em conta as diferentes versões.
dmDriverExtra	Contém o número de bytes de dados de controlador privado que segue a estrutura *DEVMODE*. Se um controlador de dispositivo não usa informações específicas do dispositivo, deixe esse membro como zero.
dmFields	Especifica qual dos membros restantes na estrutura *DEVMODE* foi inicializada. O bit zero (definido como *DM_ORIENTATION*) corresponde a *dmOrientation*; o bit 1 (definido como *DM_PAPERSIZE*), que especifica *dmPaperSize*, e assim por diante. Um controlador de impressora suporta somente aqueles membros que são apropriados para a tecnologia da impressora.
dmOrientation	Seleciona a orientação do papel Este membro pode ser *DMORIENT_PORTRAIT* (1) ou *DMORIENT_LANDSCAPE* (2).
dmPaperSize	Seleciona o tamanho do papel onde imprimir. Você pode definir esse membro como zero se o tamanho e a largura do papel forem ambos definidos pelos membros *dmPaperLength* e *dmPaperWidth*. Caso contrário, pode definir o membro *dmPaperSize* para um dos valores predefinidos listados pela Tabela 1420.3.
dwPaperLength	Anula o comprimento do papel que o membro *dmPaperSize* especifica, ou para tamanhos de papel personalizados ou para dispositivos, tais como impressoras de matriz de pontos, que podem imprimir em uma parâmetro de tamanho arbitrário Esse valor, juntamente com todos os outros nessa estrutura que especificam um comprimento físico, estão em décimos de milímetro.
dmPaperWidth	Anula a largura do papel que o membro *dmPaperSize* especifica.
dmScale	Especifica o fator pelo qual a estrutura de dados *DEVMODE* deve escalar a saída impressa. Esse membro escala o tamanho de página aparente a partir do tamanho de página físico por um fator de *dmScale/100*. Por exemplo, uma página de tamanho carta com um valor *dmScale* de 50 conteria tantos dados quanto uma página de 17x22 polegadas porque o texto de saída e os gráficos teriam a metade de sua largura e altura originais.
dmCopies	Seleciona o número de cópias impressas se o dispositivo suporta cópias em múltiplas páginas.

Tabela 1420.2 Os membros da estrutura ***DEVMODE.*** (*Continuação*)

Membros	Descrição
dmDefaultSource	Reservado — precisa ser zero.
dmPrintQuality	Especifica a resolução da impressora. Existem quatro valores independentes do dispositivo predefinidos: *DMRES_HIGH* *DMRES_MEDIUM* *DMRES_LOW* *DMRES_DRAFT* Se esse membro contiver um valor positivo não-constante, o valor especificará o número de pontos por polegada (DPI) e, portanto, dependerá do dispositivo.
dmColor	Alterna entre colorido e monocromático em impressoras coloridas. Os valores seguintes são *dmColor* possíveis: *DMCOLOR_COLOR* *DMCOLOR_MONOCHROME*
dmDuplex	Seleciona impressão dúplex, ou nos dois lados, para as impressoras com essa capacidade. Os valores seguintes são *dmDuplex* possíveis. *DMDUP_SIMPLEX* *DMDUP_HORIZONTAL* *DMDUP_VERTICAL*
dmYResolution	Especifica a resolução y, em pontos por polegada, da impressora. Se a impressora inicializa esse membro, o membro *dmPrintQuality* especifica a resolução x, em pontos por polegada, da impressora.
dmTTOption	Especifica como a impressora deve imprimir as fontes *TrueType*®. Esse membro pode ser um dos seguintes valores: *DMTT_BITMAP* Imprime as fontes *TrueType* como gráficos. Esta é a ação padrão para as impressoras de matriz de ponto. *DMTT_DOWNLOAD* Descarrega as fontes *TrueType* como fontes de software. Esta é a ação padrão para as impressoras Hewlett-Packard que usam a linguagem PCL (Printer Control Language). *DMTT_SUBDEV* Substitui as fontes de dispositivo por fontes *TrueType*. Esta é a ação padrão para as impressoras *PostScript*®.
dmCollate	Especifica se a impressora deve usar agrupamento ao imprimir múltiplas cópias. Usar *DMCOLLATE_FALSE* propicia uma impressão mais rápida e eficiente porque envia os dados para a impressora apenas uma vez, independentemente de quantas cópias você precise. O programa simplesmente diz à impressora para imprimir a página novamente. O membro *dmCollate* pode ter um dos seguintes valores: *DMCOLLATE_TRUE* Agrupa ao imprimir múltiplas cópias. *DMCOLLATE_FALSE* Não agrupa ao imprimir múltiplas cópias.
dmFormName	Especifica o nome do formulário a usar (por exemplo, Carta ou Ofício). Você pode obter o conjunto completo de nomes por meio da função *EnumForms* do Windows.
dmUnusedPadding	Alinha a estrutura em um limite de *DWORD*. Não use ou referencie esse membro. O uso é reservado para a Microsoft, que poderá mudá-lo em versões futuras do Windows.
dmBitsPerPel	Especifica em bits por pixel a resolução de cores do dispositivo de tela. Por exemplo, 4 bits para 16 cores, 8 bits para 256 cores, ou 16 bits para 65.536 cores.
dmPelsWidth	Especifica a largura, em pixels, da superfície visível do dispositivo.

Tabela 1420.2 Os membros da estrutura ***DEVMODE***. (*Continuação*)

Membros	Descrição
dmPelsHeight	Especifica a altura, em pixels, da superfície visível do dispositivo.
dmDisplayFlags	Especifica o modo de exibição do dispositivo. Os seguintes são sinalizadores válidos para o membro *dmDisplayFlags:* *DM_GRAYSCALE* Especifica que a exibição é um dispositivo não-colorido. Se você não definir esse sinalizador, o programa assumirá colorido. DM_INTERLACED Especifica que o modo de exibição é entrelaçado. Se você não definir esse sinalizador, o programa assumirá que o modo de exibição não é entrelaçado.
dmDisplayFrequency	Especifica a freqüência, em Hertz (ciclos por segundo), do dispositivo de exibição em um determinado modo.

Como a Tabela 1420.2 indica, o parâmetro *dmPaperSize* aceita constantes predefinidas que correspondem aos tamanhos de papel comumente usado internacionalmente. A Tabela 1420.3 lista algumas das constantes possíveis para o tamanho do papel.

Tabela 1420.3 Alguns valores possíveis para o tamanho de papel.

Valor	Dimensões do Papel
DMPAPER_LETTER	Carta, 8,5x11 polegadas
DMPAPER_LEGAL	Ofício, 8,5x14 polegadas
DMPAPER_A4	Folha A4, 210x297 milímetros
DMPAPER_LEDGER	Sulfite, 17x11 polegadas
DMPAPER_STATEMENT	5,5x8,5 polegadas
DMPAPER_EXECUTIVE	7,25x10,5 polegadas
DMPAPER_FOLIO	8,5x13 polegadas
DMPAPER_QUARTO	215x275 milímetros
DMPAPER_11X17	Folha de 11x17 polegadas
DMPAPER_ENV_10	Envelope #10, 4,25x9,5 polegadas
DMPAPER_FANFOLD_US	Americano padrão, 14 7/8x11 polegadas
DMPAPER_FANFOLD_LGL_GERMAN	Ofício alemão, 8,5x13 polegadas

Os dados privados de um controlador de dispositivo seguirão o membro *dmDisplayMode*. O membro *dmDriverExtra* especifica o número de bytes de dados privados. Os aplicativos escritos para as versões anteriores do Windows usavam o parâmetro *lpszSaida* para especificar um nome de porta ou para imprimir para um arquivo. Os aplicativos baseados em Win32 não precisam especificar um nome de porta; podem imprimir em um arquivo chamando a função *StartDoc* com uma estrutura *DOCINFO* cujo parâmetro *lpszSaida* especifica o caminho de um nome de arquivo de saída. Quando você não precisar mais do dispositivo do contexto, chame a função *DeleteDC* para removê-lo.

Para compreender melhor o processamento que *CreateDC* executa, considere o programa *Print_File.cpp* contido no CD-ROM que acompanha este livro. Ele imprime uma única linha de texto na impressora quando o usuário seleciona o item de menu *Testar!*. *CreateDC* cria o dispositivo de contexto para a impressora. Como o código do programa indica, o nome da impressora é "HP LaserJet 4 Plus", mas você pode alterá-lo para ficar igual ao controlador para o seu sistema, ou usar a função *EnumPrinters* para determinar qual impressora você terá conectada ao seu computador. A função *WndProc* contém o processamento operativo para o programa *Print_File.cpp*.

1421 USANDO CREATECOMPATIBLEDC PARA CRIAR UM DISPOSITIVO DO CONTEXTO EM MEMÓRIA

Em dicas anteriores você aprendeu como seus programas podem usar dispositivos de contexto para gerar saída tanto no vídeo quanto na impressora. No entanto, seus programas geralmente não desenham diretamente em um dispositivo do contexto. A função *CreateCompatibleDC* criará um dispositivo do contexto (DC) em memória compatível com um dispositivo que você especificar. Antes de um aplicativo poder usar um dispositivo do contexto em memória para as operações de desenho, você precisa selecionar um mapa de bits da largura e altura corretas no dispositivo do contexto. Após você ter selecionado um mapa de bits, poderá usar o dispositivo do contexto para preparar imagens que um programa copiará na tela ou que imprimirá. Sempre que seus programas trabalharem com mapas de bits (sobre os quais você aprenderá em dicas posteriores), seus programas colocarão o mapa de bits em um dispositivo do contexto em memória primeiro, depois o copiará para um dispositivo do contexto específico. Seus programas usarão a função *CreateCompatibleDC* como mostrado no protótipo a seguir:

```
HDC  CreateCompatibleDC(HDC hdc);
```

O parâmetro *hdc* identifica o dispositivo do contexto. Se o indicativo *hdc* for *NULL*, a função *CreateCompatibleDC* criará um dispositivo do contexto em memória compatível com a tela atual do aplicativo. Se *CreateCompatibleDC* for bem-sucedido, o valor de retorno será um indicativo para um dispositivo do contexto em memória. Se *CreateCompatibleDC* falhar, o valor de retorno será *NULL*.

Você pode usar a função *CreateCompatibleDC* somente com dispositivos que suportam as operações de varredura de bits (*raster*). Um aplicativo pode determinar se um dispositivo suporta operações de varredura chamando a função *GetDeviceCaps*. Quando você não precisar mais do dispositivo do contexto em memória, chame a função *DeleteDC* para excluí-lo.

Para compreender melhor o processamento que a função *CreateCompatibleDC* executa, considere o programa *Draw_Bitmap.cpp* contido no CD-ROM que acompanha este livro. O programa *Draw_Bitmap* carrega um mapa de bits e o coloca em um dispositivo do contexto em memória, depois o copia na área cliente da janela. Como é normal, a função *WndProc* contém o código de processamento operativo do programa *Draw_Bitmap*.

1422 COMPREENDENDO OS PERIGOS DE CREATEDC

Como você aprendeu na Dica 1420, seus programas geralmente usam a função *CreateDC* para obter um dispositivo de contexto para uma impressora. No entanto, seus programas também podem usar *CreateDC* para obter o dispositivo do contexto para a tela do monitor (a tela de hardware, não a área cliente de uma única janela, ou a área cliente da área de trabalho). Quando você usa *CreateDC* para obter o dispositivo do contexto para a tela, seus programas podem, na verdade, desenhar em qualquer lugar na tela, não apenas dentro das áreas limites de um programa. Além de potencialmente produzir resultados imprevisíveis, esse processamento não é consistente com o padrão do Windows. Ao tentar obter um dispositivo do contexto para uma janela na tela, seus programas sempre devem usar a função *GetDC* ou *BeginPaint*, não a função *CreateDC*.

1423 USANDO A FUNÇÃO CREATEFONT

À medida que seus programas trabalham com dispositivos do contexto e criam muitas telas interessantes, você pode achar que eles precisam modificar as fontes que uma janela exibe, ou criar fontes personalizadas. Seus programas podem criar fontes usando a função *CreateFont* ou *CreateFontIndirect*. Se você estiver manipulando múltiplas fontes, geralmente deverá usar a função *CreateFontIndirect*. Se você estiver manipulando apenas uma única fonte, em geral deverá usar a função *CreateFont*. A função *CreateFont* permite que seus programas criem uma fonte lógica (uma definição de fonte numérica) com características específicas. Subsequentemente, você pode selecionar a fonte lógica como a fonte para qualquer dispositivo. Seus programas usarão a função *CreateFont*, como segue:

```
HFONT CreateFont(
   int hAltura,      // altura lógica da fonte
   int nLarg,        // largura lógica média do caractere
   int nEscap,       // ângulo de escapement
   int nOrient,      // ângulo de orientação da linha-base
```

```
    int fnPeso,       // peso da fonte
    DWORD fdwItalico,// sinalizador de atributo itálico
    DWORD fdwSubli,   // sinalizador de atributo sublinhado
    DWORD fdwRiscado,// sinalizador de atributo riscado
    DWORD fdwConjCarac,       // identificador do conjunto de caractere
    DWORD fdwPrecSaida,       // precisão da saída
    DWORD fdwPrecCorte,       // precisão do corte
    DWORD fdwQualid,          // qualidade da saída
    DWORD fdwPitchEFamilia,   // pitch e família
    LPCTSTR lpszFace          // ponteiro para string de nome da fonte
);
```

A função *CreateFont* aceita os parâmetros detalhados na Tabela 1423.1.

Tabela 1423.1 Os parâmetros para a função ***CreateFont***.

Parâmetro	Descrição
nAltura	Especifica a altura, em unidades lógicas, da célula de caracteres da fonte ou caractere. O valor da altura do caractere (também conhecido como a altura em) é o valor da altura da célula do caractere menos o valor do *leading* interno. O mapeador de fonte interpreta o valor especificado em *nAltura*. Se o valor para *nAltura* for maior que zero, o mapeador de fonte transformará esse valor em unidades de dispositivo e o comparará com a altura da célula das fontes disponíveis. Se o valor para *nAltura* for igual a zero, o mapeador de fonte usará um valor de altura padrão ao procurar uma ocorrência. Se o valor para *nAltura* for menor que zero, o mapeador de fonte transformará esse valor em unidades de dispositivo, e comparará seu valor absoluto com a altura do caractere das fontes disponíveis Para todas as comparações de altura, o mapeador de fonte procura a maior fonte que não excede o tamanho solicitado. Esse mapeamento ocorre quando você usa a fonte pela primeira vez
nLargura	Especifica a largura média, em unidades lógicas, dos caracteres na fonte solicitada Se o valor for zero, o mapeador de fonte escolherá um valor "ocorrência mais próxima". O parâmetro *nLargura* determina o valor "ocorrência mais próxima" comparando os valores absolutos da diferença entre a relação altura/largura e a relação digitalizada das fontes disponíveis.
nEscap	Especifica o ângulo, em décimos de grau, entre o vetor *escapement* e o eixo-x do dispositivo. O vetor de *escapement* é paralelo à linha-base de uma linha de texto. Sob o Windows NT, quando você definir o modo gráfico como *GM_ADVANCED*, poderá especificar o ângulo de *escapement* da string, independente do ângulo de orientação dos caracteres da string. Quando você definir o modo gráfico para *GM_COMPATIBLE*, *nEscap* especificará tanto o vetor de *escapement* quanto o ângulo de orientação. Em geral, você deve definir *nEscap* e *nOrientation* com o mesmo valor.
nOrient	Especifica o ângulo, em décimos de grau, entre a linha-base de cada caractere e o eixo-x do dispositivo.
fnPeso	Especifica o peso da fonte no intervalo de 0 até 1000. Por exemplo, 400 é normal e 700 é negrito. Se o valor for 0, o sistema operacional usa um peso padrão. O parâmetro *fnPeso* pode ser um dos valores que a Tabela 1423.2 listará a seguir.
fdwItalico	Especifica uma fonte itálica se definido como *True*.
fdwSubli	Especifica uma fonte sublinhada se definido como *True*.
fdwRiscado	Especifica uma fonte riscada se definido como *True*

Tabela 1423.1 Os parâmetros para a função *CreateFont*. (*Continuação*)

Parâmetro	Descrição
fdwConjCarac	Especifica o conjunto de caracteres. Os valores na Tabela 1423.3, logo à frente, são predefinidos. O valor *OEM_CHARSET* especifica um conjunto de caracteres que é dependente do sistema operacional. Você pode usar o valor *DEFAULT_CHARSET* para permitir que o nome e o tamanho de uma fonte descrevam plenamente a fonte lógica. Se o nome da fonte especificada não existir, o sistema operacional poderá substituir uma fonte de qualquer conjunto de caracteres pela fonte especificada; portanto, você deve usar *DEFAULT_CHARSET* de forma frugal, para evitar resultados inesperados. Fontes com outros conjuntos de caracteres podem existir no sistema operacional. Se um aplicativo usa uma fonte com um conjunto de caracteres desconhecido, ele não deve tentar traduzir ou interpretar as strings que ele produz com essa fonte. O parâmetro *fdwConjCarac* é importante no processo de mapeamento de fonte. Para garantir resultados consistentes, você precisa especificar um determinado conjunto de caracteres. Se você especificar um nome de fonte no parâmetro *lpszFace*, certifique-se de que *fdwConjCaract* corresponde ao conjunto de caractere da fonte especificada em *lpszFace*.
fdwPrecSaida	Especifica a precisão da saída. A *precisão da saída* define como a saída precisa corresponder à altura, à largura, à orientação de caractere, o ângulo, pitch e tipo de fonte. Pode ser um dos valores que a Tabela 1423.4 detalha. Os aplicativos podem usar os valores *OUT_DEVICE_PRECIS*, *OUT_RASTER_PRECIS* e *OUT_TT_PRECIS* para controlar como o mapeador de fonte escolhe uma fonte quando o sistema operacional o contém mais do que uma fonte com um nome especificado. Por exemplo, se um sistema operacional contém uma fonte chamada Symbol na forma *TrueType* e de varredura. Especificar *OUT_TT_PRECIS* força o mapeador de fonte a escolher uma fonte *TrueType*, mesmo se ele precisar substituir uma fonte *TrueType* de outro nome.
fdwPrecCorte	Especifica a precisão de corte. A *precisão de corte* define como cortar os caracteres que estão parcialmente fora da região de corte. Ela pode ser um dos valores que a Tabela 1423.5 detalhará mais à frente.
fdwQualid	Especifica a *qualidade da saída*, que define como a interface de dispositivos gráficos precisa tentar casar os atributos de fonte lógica com os de uma fonte física real. Pode ser um dos valores listados na Tabela 1423.6, à frente.
fdwPitchEFamilia	Especifica o pitch e a família da fonte. Os dois bits menos significativos especificam o pitch da fonte, e podem ser um dos seguintes valores: *DEFAULT_PITCH* *FIXED_PITCH* *VARIABLE_PITCH* Os quatro bits mais significativos especificam a família da fonte, e podem ser um dos valores listados na Tabela 1423.7, mais à frente. Um aplicativo pode especificar um valor para o parâmetro *fdwPitchEFamilia* usando o operador Booleano *OU* para juntar uma constante de pitch com uma de família. As famílias de fontes descrevem o aspecto de uma fonte de um modo geral. Elas especificarão as fontes quando a fonte exata solicitada não estiver disponível.
lpszFace	Aponta para uma string terminada por *NULL* que especifica o nome da fonte. O comprimento dessa string não deve exceder 32 caracteres, incluindo o terminador *NULL*. Você pode usar a função *EnumFontFamilies* para enumerar os nomes de todas as fontes disponíveis. Se *lpszFace* for *NULL* ou apontar para uma string vazia, a interface de dispositivo gráfico usará a primeira fonte que for igual aos outros atributos especificados.

Como você aprendeu na Tabela 1423.1, seus programas podem definir uma variedade de "pesos" de fonte predefinidos ao criar uma fonte lógica. A Tabela 1423.2 detalha os possíveis valores para o parâmetro *nPeso*.

Tabela 1423.2 Os possíveis valores para o parâmetro ***nPeso***.

Valor	Peso	Valor	Peso
FW_DONTCARE	0	*FW_SEMIBOLD*	600
FW_THIN	100	*FW_BOLD*	700
FW_EXTRALIGHT	200	*FW_EXTRABOLD*	800
FW_LIGHT	300	*FW_ULTRABOLD*	800
FW_NORMAL	400	*FW_HEAVY*	900
FW_REGULAR	400	*FW_BLACK*	900
FW_MEDIUM	500		

Como você também viu na Tabela 1423 1, a função *CreateFont* lhe permite especificar um conjunto de caracteres predefinidos para a fonte. A Tabela 1423.3 lista os valores de constante possíveis para o conjunto de caracteres.

Tabela 1423.3 Constantes de conjuntos de caracteres predefinidos.

Conjuntos de Caracteres Predefinidos		
ANSI_CHARSET	*DEFAULT_CHARSET*	*SYMBOL_CHARSET*
SHIFTJIS_CHARSET	*GB2312_CHARSET*	*HANGEUL_CHARSET*
CHINESEBIG5_CHARSET	*OEM_CHARSET*	
Windows 95 somente		
JOHAB_CHARSET	*HEBREW_CHARSET*	*ARABIC_CHARSET*
GREEK_CHARSET	*TURKISH_CHARSET*	*THAI_CHARSET*
EASTEUROPE_CHARSET	*RUSSIAN_CHARSET*	
MAC_CHARSET	*BALTIC_CHARSET*	

Como você aprendeu, a precisão da saída define quão fielmente a saída precisa corresponder com a altura, largura e outras características da fonte solicitada. Seus programas podem usar as constantes para o parâmetro *fdwPrecSaida* listadas na Tabela 1423.4 para controlar a precisão da saída.

Tabela 1423.4 Os valores possíveis para o parâmetro ***fdwPrecSaida***.

Valor	Significado
OUT_CHARACTER_PRECIS	Não usado.
OUT_DEFAULT_PRECIS	Especifica o comportamento do mapeador de fonte padrão.
OUT_DEVICE_PRECIS	Instrui o mapeador de fonte a escolher uma fonte Device (Dispositivo) quando o sistema contém múltiplas fontes com o mesmo nome.
OUT_OUTLINE_PRECIS	Sob o Windows NT, este valor instrui o mapeador de fonte a escolher de *TrueType* e de outras fontes baseadas em contorno. O Windows 95 não usa esse valor.
OUT_RASTER_PRECIS	Instrui o mapeador de fonte a escolher uma fonte de varredura quando o sistema contém múltiplas fontes com o mesmo nome.
OUT_STRING_PRECIS	O mapeador de fonte não usa este valor, mas a função o retorna quando o programa enumera as fontes de varredura.
OUT_STROKE_PRECIS	Sob o Windows NT, o mapeador de fonte não usa este valor, mas a função o retorna quando o programa enumera *TrueType*, outras fontes baseadas em contorno e as fontes vetoriais. O Windows 95 usa este valor para mapear as fontes vetoriais, e a função retorna o valor quando o programa enumera fontes *TrueType* ou vetoriais.

Tabela 1423.4 Os valores possíveis para o parâmetro ***fdwPrecSaida***. (*Continuação*)

Valor	Significado
OUT_TT_ONLY_PRECIS	Instrui o mapeador de fonte a escolher a partir de somente fontes *TrueType*. Se não houver fontes *TrueType* instaladas no sistema, o mapeador de fonte retornará para o comportamento padrão.
OUT_TT_PRECIS	Instrui o mapeador de fonte a escolher uma fonte *TrueType* quando o sistema contém múltiplas fontes com o mesmo nome.

Exatamente como seus programas podem controlar quão precisamente o Windows deve desenhar fontes no dispositivo do contexto, assim também seus programas podem controlar a precisão com que o Windows deve cortar os caracteres que estao parcialmente fora da região de corte. A Tabela 1423.5 detalha os valores possíveis para o parâmetro *fdwPresCorte*.

Tabela 1423.5 Os valores possíveis para o parâmetro ***fdwPresCorte***.

Valor	Significado
CLIP_DEFAULT_PRECIS	Especifica o comportamento de corte padrão.
CLIP CHARACTER PRECIS	Não usado.
CLIP_STROKE_PRECIS	O mapeador de fonte não usa esse valor, mas a função o retorna quando o programa enumera fontes de varredura, vetorias ou *TrueType*. Sob o Windows NT, por questões de compatibilidade, a função sempre retorna esse valor ao enumerar as fontes.
CLIP MASK	Não usado
CLIP_EMBEDDED	Você precisa especificar esse sinalizador para usar uma fonte incorporada de leitura somente.
CLIP_LH_ANGLES	Quando vôcê usa esse valor, a rotação para todas as fontes depende se a orientação do sistema de coordenadas é para a esquerda ou para a direita. Se você não usar esse valor, as fontes de dispositivo sempre serão rotacionadas em sentido anti-horário, mas a rotação das outras fontes será dependente da orientação do sistema de coordenadas
CLIP_TT_ALWAYS	Não usado.

Além da precisão da saída, seus programas podem controlar a qualidade da saída. Existem três qualidades de saída possíveis que o parâmetro *fdwQualid* usa para as suas fontes, como detalha a Tabela 1423.6.

Tabela 1423.6 Os valores possíveis para o parâmetro ***fwdQualid***.

Valor	Significado
DEFAULT_QUALITY	A aparência da fonte não importa.
DRAFT_QUALITY	A aparência da fonte é menos importante do que quando você usa o valor *PROOF_QUALITY*. Para as fontes de varredura da interface de dispositivo gráfico, o sistema operacional habilita o ajuste da escala, o que significa que mais tamanhos de fonte estão disponíveis, mas a qualidade pode ser menor. O sistema operacional sintetiza as fontes negrito, itálico, sublinhado e riscado, se necessário
PROOF_QUALITY	A qualidade do caractere da fonte é mais importante do que o casamento exato dos atributos de fonte lógica. Para as fontes de varredura da interface de dispositivo gráfico, o ajuste da escala está desabilitado, e o sistema operacional escolhe a fonte mais próxima em tamanho. Embora o tamanho da fonte escolhido possa não estar mapeado exatamente quando você usar *PROOF_QUALITY*, a qualidade da fonte é alta e não há distorção da aparência O sistema operacional sintetiza as fontes negrito, itálico, sublinhado e riscado, se necessário.

Finalmente, seus programas podem especificar o pitch e a família de uma fonte, o que ajuda a função *CreateFont* a chegar mais perto da fonte, se não encontrar uma correspondência exata. A Tabela 1423.7 detalha as famílias de fontes que você pode usar com o parâmetro *dwPitchEFamilia*.

Tabela 1423.7 Os valores de família possíveis para o parâmetro ***fdwPitchEFamilia***

Valor	Descrição
FF_DECORATIVE	Fontes novas. Um exemplo é Old English.
FF_DONTCARE	Seu programa não quer saber ou não conhece.
FF_MODERN	Fontes com larguras de traço constantes, com ou sem serifas. Exemplos são Pica, Elite e Courier New®.
FF_ROMAN	Fontes com largura de traço variável e com serifas. MS Serif® é um exemplo.
FF_SCRIPT	Fontes projetadas para ter a aparência de escrita à mão. Script e Cursive são exemplos.
FF_SWISS	Fontes com largura de traço variável e sem serifas. MS Sans Serif é um exemplo.

Se a função *CreateFont* for bem-sucedida, o valor de retorno será um indicativo para uma fonte lógica. Se a função *CreateFont* falhar, o valor de retorno será *NULL*. Quando você não precisar mais da fonte, chame a função *DeleteObject* para excluí-la.

Para ajudar a proteger os direitos autorais dos criadores que fornecem fontes para o sistema operacional Windows, os aplicativos sempre devem informar o nome exato de uma fonte selecionada. Como as fontes disponíveis podem variar de um sistema para outro, não assuma que a fonte selecionada é sempre a mesma que a fonte solicitada. Por exemplo, se você solicitar uma fonte chamada "Palatino", mas essa fonte não estiver disponível no sistema, o mapeador de fonte substituirá uma fonte com atributos similares mas um nome diferente. Sempre informe o nome da fonte selecionada para o usuário.

Para compreender melhor o processamento que a função *CreateFont* executa, considere o programa *Create_BigRoman*, contido no CD-ROM que acompanha este livro. Esse programa cria uma fonte Times New Roman que tem um tamanho de 21 por 16 unidades. O programa então usa a fonte recém-criada para pintar texto na área cliente da janela. A função *WndProc* contém o processamento operativo do programa *Create_BigRoman*.

USANDO A FUNÇÃO ENUMFONTFAMILIES 1424

Como você aprendeu na dica anterior, o Windows agrupa as fontes juntas em famílias, com base em uma série de características comuns a cada fonte dentro da família. Seus programas podem usar a função *EnumFontFamilies* para enumerar em um família de fonte especificada as fontes disponíveis em um dispositivo especificado. Dentro de seus programas, você geralmente usará a função *EnumFontFamilies* para determinar quais fontes você pode usar com um dispositivo específico e para obter um ponteiro para uma estrutura *LOGFONT* que seus programas podem usar junto com a função *CreateFontIndirect* para criar a fonte para o dispositivo especificado. Seus programas usarão a função *EnumFontFamilies*, como segue:

```
int EnumFontFamilies(
   HDC hdc,                          // indicativo para o dispositivo
                                     // do contexto
   LPCTSTR  lpszFamilia,             // ponteiro para string nome-família
   FONTNUMPROC  lpEnumFontFamProc,   // ponteiro para função de callback
   LPARAM lParam                     // endereço dos dados fornecidos
                                     // pelo aplicativo
);
```

A Tabela 1424.1 detalha os parâmetros que a função ***EnumFontFamilies*** aceita.

Tabela 1424.1 Os parâmetros da função *EnumfontFamilies*.

Parâmetro	Descrição
hdc	Identifica o dispositivo do contexto.
lpszFamilia	Aponta para uma string terminada por *NULL* que especifica o nome da família das fontes desejadas. Se *lpszFamilia* for *NULL*, a função *EnumFontFamilies* selecionará aleatoriamente, e enumerará uma fonte de cada família de tipo disponível.
lpEnumFontFamProc	Especifica o endereço da ocorrência do procedimento da função de callback definida pelo aplicativo. Para informações sobre a função de callback, veja a função *EnumFontFamProc*
lParam	Aponta para dados fornecidos pelo aplicativo. A função passa os dados para a função de callback juntamente com as informações da fonte.

Se a função *EnumFontFamilies* for bem-sucedida, o valor de retorno será o último valor que a função de callback retorna. O significado desse valor depende da implementação. A função *EnumFontFamilies* recupera os nomes de estilo associados com uma fonte *TrueType*. Com *EnumFontFamilies*, você pode obter informações sobre estilos de fontes incomuns (por exemplo, Outline). Para cada fonte que tenha o nome de fonte que o parâmetro *lpszFamilia* especifica, a função *EnumFontFamilies* obtém informações sobre essa fonte e a passa para a função para a qual o parâmetro *lpEnumFontFamProc* aponta. A função de callback definida pelo aplicativo pode processar as informações de fonte conforme desejado. A enumeração continua até que não haja fontes ou até que a função de callback retorne zero. Você usará a função de callback como segue:

```
int CALLBACK EnumFontFamProc(
   ENUMLOGFONT* lpelf,     // ponteiro para uma estrutura ENUMLOGFONT
   NEWTEXTMETRIC* lpntm,   // ponteiro para estrutura NEWTEXTMETRIC
   int nFontType,          // tipo da fonte
   LPARAM lParam           // dados definidos pelo aplicativo
);
```

Como você pode ver, a função de callback aceita quatro parâmetros. O primeiro é um ponteiro para uma estrutura *ENUMLOGFONT*. A estrutura *ENUMLOGFONT* define os atributos, o nome completo e o estilo de uma fonte. A API Win32 define a estrutura *ENUMLOGFONT* como segue:

```
typedef struct tagENUMLOGFONT  {
   LOGFONT    elfLogFont;
   BCHAR      elfFullName[LF_FULLFACESIZE];
   BCHAR      elfStyle[LF_FACESIZE];
 } ENUMLOGFONT;
```

A Tabela 1424.2 define os membros da estrutura *ENUMLOGFONT*.

Tabela 1424.2 Os membros da estrutura *ENUMLOGFONT*.

Membro	Descrição
elfLogFont	Especifica uma estrutura *LOGFONT* que define os atributos de uma fonte.
elfFullName	Especifica um nome único para a fonte. Por exemplo "Companhia de Fontes ABCD TrueType Bold Italic Sans Serif".
elfStyle	Especifica o estilo da fonte. Por exemplo, "Bold Italic". A estrutura *TEXTMETRIC* contém informações básicas sobre uma fonte física. Todos os tamanhos são dados em unidades lógicas; isto é, dependem do modo de mapeamento atual do conteúdo da tela.

Os outros três parâmetros especificam informações específicas de *TrueType* e o tipo da fonte (ou *DEVICE_FONTTYPE, RASTER_FONTTYPE,* ou *TRUETYPE_FONTTYPE,* ou alguma combinação dos três).

Para compreender melhor o processamento que a função *EnumFontFamilies* executa, considere o programa *Enum_AllFonts.cpp,* contido no CD-ROM que acompanha este livro. O programa usa a função *EnumFontFamilies* para preencher uma caixa de lista com os nomes de fonte *TrueType* disponíveis. Quando o usuário seleciona o item de menu *Testar!,* o programa exibe uma string simples de teste no formato de fonte selecionado no momento. O processamento operativo do programa *Enum_AllFonts.cpp* ocorre dentro das funções *WndProc, EnumFontProc* e *FindFontProc.*

EXIBINDO MÚLTIPLAS FONTES COM CREATEFONTINDIRECT 1425

Como você aprendeu, seus programas podem usar a função *CreateFont* para criar fontes. No entanto, o número de parâmetros para uma única chamada à *CreateFont* é suficiente para tornar *CreateFont* difícil de usar. Uma alternativa melhor é a função *CreateFontIndirect.* A função *CreateFontIndirect* cria uma fonte lógica que tem as características especificadas em uma determinada estrutura. Você pode subseqüentemente selecionar a fonte com a fonte atual para qualquer dispositivo do contexto, como mostrado aqui:

```
HFONT CreateFontIndirect(CONST LOGFONT *lplf);
```

O parâmetro *lplf* aponta para uma estrutura que define as características da fonte lógica. A estrutura *LOGFONT* define os atributos de uma fonte, como mostrado aqui:

```
typedef struct tagLOGFONT {
   LONG lfHeight;
   LONG lfWidth;
   LONG lfEscapement;
   LONG lfOrientation;
   LONG lfWeight;
   BYTE lfItalic;
   BYTE lfUnderline;
   BYTE lfStrikeOut;
   BYTE lfCharSet;
   BYTE lpOutPrecision;
   BYTE lpClipPrecision;
   BYTE lfQuality;
   BYTE lfPitchAndFamily;
   TCHAR lfFaceName[LF_FACESIZE];
 } LOGFONT;
```

Se você olhar atentamente para a estrutura *LOGFONT,* verá que seus membros correspondem diretamente aos parâmetros para a função *CreateFont* na Dica 1423. Na verdade, os membros aceitam os mesmos valores que os parâmetros para a função *CreateFont.*

Se a função *CreateFontIndirect* for bem-sucedida, o valor de retorno será um indicativo para uma fonte lógica. Se a função *CreateFontIndirect* falhar, o valor de retorno será *NULL.* A função *CreateFontIndirect* cria uma fonte lógica com as características especificadas na estrutura *LOGFONT.* Quando você selecionar a fonte com a função *SelectObject,* o mapeador da fonte da interface de dispositivo gráfico tentará casar a fonte lógica com uma fonte física existente. Se a interface de dispositivo gráfico não encontar uma ocorrência exata, ela fornecerá uma alternativa cujas características combinam com o maior número possível de características requisitadas. Quando você não precisar mais da fonte, chame a função *DeleteObject* para excluí-la.

O programa *Enum_AllFonts.cpp* apresentado na dica anterior usou a função *CreateFontIndirect.* Em vez de chamar *CreateFont* com quatroze parâmetros, o programa chama *CreateFontIndirect* com somente um único parâmetro, como mostrado aqui:

```
hFont = CreateFontIndirect(&lf);
```

1426 RECUPERANDO AS CAPACIDADES DE UM DISPOSITIVO

Como você aprendeu, seus programas usarão dispositivos do contexto para gerenciar texto e gráficos dentro de seus aplicativos, tanto para exibições na tela quanto para saída em dispositivos, tais como impressoras ou plotters. Quando seu programa precisar gerar saída em uma impressora ou em um plotter, você geralmente irá requerer informações sobre as capacidades do dispositivo antes de escrever ou desenhar no dispositivo. Dentro de seus programas, você poderá usar a função *GetDeviceCaps* para recuperar informações específicas do dispositivo sobre um dispositivo especificado. Você usará *GetDeviceCaps* dentro de seus programas, como mostrado no protótipo a seguir:

```
int GetDeviceCaps(HDC hdc, int nIndice);
```

O parâmetro *hdc* identifica o dispositivo do contexto. O parâmetro *nIndice* especifica o item a retornar. O parâmetro *nIndice* pode ser um dos valores detalhados na Tabela 1426.1.

Tabela 1426.1 Os valores possíveis para o parâmetro ***nIndice***.

Índice	Significado
DRIVERVERSION	Versão do controlador de dispositivo
TECHNOLOGY	Tecnologia do dispositivo. Pode ser qualquer um dos valores detalhados na Tabela 1426.2.
HORZSIZE	Largura, em milímetros, da tela física.
VERTSIZE	Altura, em milímetros, da tela física.
HORZRES	Largura, em pixels, da tela.
VERTRES	Altura, em linhas de varredura, da tela.
LOGPIXELSX	Número de pixels por polegada lógica juntamente com a largura da tela.
LOGPIXELSY	Número de pixels por polegada lógica juntamente com a largura da tela.
BITSPIXEL	Número de bits de cores adjacentes para cada pixel.
PLANES	Número de planos de cores.
NUMBRUSHES	Número de pincéis específicos do dispositivo.
NUMPENS	Número de canetas específicas do dispositivo.
NUMFONTS	Número de fontes específicas do dispositivo.
NUMCOLORS	Número de entradas na tabela de cores do dispositivo, se o dispositivo tiver uma profundidade de cores de não mais de 8 pixels por dispositivo. Para os dispositivos com profundidades de cor maiores, *GetDeviceCaps* retorna -1.
ASPECTX	Largura relativa de um pixel do dispositivo que o dispositivo usa para o desenho de linhas.
ASPECTY	Altura relativa de um pixel do dispositivo que o dispositivo usa para o desenho de linhas.
ASPECTXY	Largura diagonal do pixel do dispositivo que o dispositivo usa para o desenho de linhas.
PDEVICESIZE	Reservado.
CLIPCAPS	Sinalizador que indica as capacidades de corte do dispositivo. Se o dispositivo puder cortar em um retângulo, o valor será 1; caso contrário, será 0.

Tabela 1426.1 Os valores possíveis para o parâmetro ***nIndice***. (*Continuação*)

Índice	Significado
SIZEPALETTE	Número de itens na paleta do sistema. Esse índice é válido somente se o controlador de dispositivo definir o bit *RC_PALETTE* no índice *RASTERCAPS*, e estará disponível somente se o controlador for compatível com o Windows versão 3.0 ou posterior.
NUMRESERVED	Número de itens reservados na paleta do sistema. Esse índice é válido somente se o controlador de dispositivo definir o bit *RC_PALETTE* no índice *RASTERCAPS*, e estará disponível somente se o controlador for compatível com o Windows versão 3.0 ou posterior.
COLORRES	Resolução de cor real do dispositivo, em bits por pixel. Este índice será válido somente se o controlador de dispositivo definir o bit *RC_PALETTE* no índice *RASTERCAPS*, e estará disponível somente se o controlador for compatível com o Windows versão 3.0 ou posterior.
PHYSICALWIDTH	Para os dispositivos de impressão, a largura da página física, em unidades de dispositivo. Por exemplo, uma impressora configurada para imprimir em 600 dpi em papel 8x11 tem um valor largura de 5100 unidades de dispositivos. Observe que a página física é quase sempre maior que a área imprimível da página e nunca menor.
PHYSICALHEIGHT	Para os dispositivos de impressão, a largura da página física, em unidades de dispositivo. Por exemplo, uma impressora configurada para imprimir em 600 dpi em papel 8x11 tem um valor de altura de 6600 unidades de dispositivos. Observe que a página física é quase sempre maior que a área imprimível da página e nunca menor.
PHYSICALOFFSETX	Para os dispositivos de impressão, a distância da borda esquerda da página física até a margem esquerda da área de impressão, em unidades de dispositivo. Por exemplo, uma impressora configurada para imprimir em 600 dpi em papel 8x11, que não pode imprimir na 0,25" esquerda do papel tem um deslocamento físico horizontal de 150 unidades de dispositivo.
PHYSICALOFFSETY	Para os dispositivos de impressão, a distância da borda superior da página física até o topo da área de impressão, em unidades de dispositivo. Por exemplo, uma impressora configurada para imprimir em 600 dpi em papel 8x11, que não pode imprimir na 0,5" superior do papel tem um deslocamento físico vertical de 300 unidades de dispositivo.
VREFRESH	Para os dispositivos de exibição, a taxa de renovação vertical atual, em ciclos por segundo (Hz), sob o Windows 95 somente Um valor de taxa de renovação vertical de 0 ou 1 representa a taxa de renovação padrão do hardware de exibição. Essa taxa padrão é tipicamente configurada por *dip-switches* na placa de vídeo ou na placa-mãe do computador ou por meio de um programa de configuração que não usa as funções de exibição do Win32, tal como *ChangeDisplaySettings*.
DESKTOPHORZRES	Largura, em pixels, da área de trabalho virtual, sob o Windows NT somente. Esse valor poderá ser maior que *HORZRES* se o dispositivo suportar uma área de trabalho virtual ou múltiplos monitores.
DESKTOPVERTRES	Altura, em pixels, da área de trabalho virtual, sob o Windows NT somente. Esse valor poderá ser maior que *VERTRES* se o dispositivo suportar uma área de trabalho virtual ou múltiplos monitores.

Tabela 1426.1 Os valores possíveis para o parâmetro ***nIndice***. (*Continuação*)

Índice	Significado
BLTALIGNMENT	O alinhamento de desenho horizontal preferido do dispositivo, expresso com um múltiplo de pixels, sob o Windows NT somente. Para melhor desempenho do desenho, seus programas devem alinhar as janelas horizontalmente em um múltiplo deste valor. O valor 0 indica que o Windows acelerou o acesso ao dispositivo, e os dispositivos do contexto para esse dispositivo podem usar qualquer alinhamento.
RASTERCAPS	Valor que indica as capacidades de varredura do dispositivo, como mostrado na Tabela 1426.3.
CURVECAPS	Valor que indica as capacidades de curva do dispositivo, como mostrado na Tabela 1426.4.
LINECAPS	Valor que indica as capacidades do dispositivo, como mostrado na Tabela 1426.5.
POLYGONALCAPS	Valor que indica as capacidades de polígono do dispositivo, como mostrado na Tabela 1426.6.
TEXTCAPS	Valor que indica as capacidades de texto do dispositivo, como mostrado na Tabela 1426.7.

Como você viu na Tabela 1426.1, se seu programa requer a tecnologia de controlador do dispositivo, a chamada *GetDeviceCaps* retorna um dos seguintes valores de sinalizador. A Tabela 1426.2 lista os valores individuais de sinalizadores.

Tabela 1426.2 Os valores de retorno para tipos possíveis de tecnologia dos controladores.

Valor	Significado
DT_PLOTTER	Plotter vetorial
DT_RASDISPLAY	Monitor de varredura
DT_RASPRINTER	Impressora de varredura
DT_RASCAMERA	Câmera de varredura
DT_CHARSTREAM	Canal de caracteres
DT_METAFILE	Meta-arquivo
DT_DISPFILE	Arquivo de exibição

É importante observar que, se o parâmetro *hdc* identificar o dispositivo do contexto de um meta-arquivo expandido, a tecnologia do dispositivo será aquela do dispositivo referenciado conforme passado anteriormente para a função *CreateEnhMetafile*. Para determinar se o contexto é um dispositivo do contexto de meta-arquivo expandido, use a função *GetObjectType*.

Finalmente, se seu programa precisar identificar as características de varredura do dispositivo em questão, sua chamada à *GetDeviceCaps* incluirá a constante *RASTERCAPS*. Quando seu programa solicita as características de varredura de um dispositivo, o valor de retorno da função é um ou mais dos valores na Tabela 1426.3.

Tabela 1426.3 Os valores de retorno possíveis para a solicitação ***RASTERCAPS***.

Capacidade	Significado
RC_BANDING	Requer suporte para o enfaixamento.
RC_BITBLT	Capaz de transferir mapas de bits.
RC_BITMAP64	Capaz de suportar mapas de bits maiores que 64K

Tabela 1426.3 Os valores de retorno possíveis para a solicitação ***RASTERCAPS***. (*Continuação*)

Capacidade	Significado
RC_DI_BITMAP	Capaz de suportar as funções *SetDIBits* e *GetDIBits*.
RC_DIBTODEV	Capaz de suportar a função *SetDIBitsToDevice*.
RC_FLOODFILL	Capaz de efetuar os preenchimentos de inundação (*flood fill*)
RC_GDI20_OUTPUT	Capaz de suportar recursos do Windows 2.0.
RC_PALETTE	Especifica um dispositivo baseado em paleta.
RC_SCALING	Capaz de ajustar a escala.
RC_STRETCHBLT	Capaz de executar a função *StretchBlt*.
RC_STRETCHDIB	Capaz de executar a função *StretchDIBits*.

Além de informações sobre as capacidades de varredura do dispositivo, seus programas normalmente irão requerer conhecimento da capacidade do dispositivo de desenhar ou exibir figuras curvas. Você pode conferir as capacidades de desenho de um dispositivo com a solicitação *CURVECAPS*. A solicitação *CURVECAPS* retorna um ou mais dos valores listados na Tabela 1426.4.

Tabela 1426.4 Os valores de retorno obtidos com a chamada à ***CURVECAPS***.

Valor	Significado
CC_NONE	O dispositivo não suporta curvas.
CC_CIRCLES	O dispositivo pode desenhar círculos.
CC_PIE	O dispositivo pode desenhar fatias de um gráfico setorial.
CC_CHORD	O dispositivo pode desenhar arcos.
CC_ELLIPSES	O dispositivo pode desenhar ovais e elipses.
CC_WIDE	O dispositivo pode desenhar bordas largas.
CC_STYLED	O dispositivo pode desenhar bordas estilizadas.
CC_WIDESTYLED	O dispositivo pode desenhar bordas que são largas e estilizadas.
CC_INTERIORS	O dispositivo pode desenhar interiores.
CC_ROUNDRECT	O dispositivo pode desenhar retângulos arredondados.

Como você aprendeu, um dispositivo pode ou não pode ter certas capacidades de desenho de curva. A maioria dos dispositivos também têm certas capacidades de desenho de linha. Seus programas podem verificar as capacidades de desenho de linha de um dispositivo com uma chamada à *LINECAPS*. A Tabela 1426.5 detalha os possíveis valores de retorno da chamada à *LINECAPS*.

Tabela *1426.5* *Os valores de retorno da chamada à* ***LINECAPS***.

Valor	Significado
LC_NONE	O dispositivo não suporta linhas.
LC_POLYLINE	O dispositivo pode desenhar uma polilinha.
LC_MARKER	O dispositivo pode desenhar um marcador.
LC_POLYMARKER	O dispositivo pode desenhar múltiplos marcadores.
LC_WIDE	O dispositivo pode desenhar linhas largas.
LC_STYLED	O dispositivo pode desenhar linhas estilizadas.
LC_WIDESTYLED	O dispositivo pode desenhar linhas que são largas e estilizadas.
LC_INTERIORS	O dispositivo pode desenhar interiores.

Além das informações sobre a capacidade de um dispositivo de desenhar linhas e círculos, seus programas normalmente irão requerer informações sobre a capacidade de um dispositivo de desenhar polígonos, incluindo retângulos, trapézios etc. Você pode conferir as capacidades de um dispositivo com uma chamada à *POLYGONALCAPS*, que retornará um ou mais dos valores na Tabela 1426.6.

Tabela 1426.6 Os valores de retorno possíveis de uma chamada a ***POLYGONALCAPS***.

Valor	Significado
PC_NONE	O dispositivo não suporta polígonos.
PC_POLYGON	O dispositivo pode desenhar polígonos de preenchimento alternado.
PC_RECTANGLE	O dispositivo pode desenhar retângulos.
PC_WINDPOLYGON	O dispositivo pode desenhar polígonos preenchidos.
PC_SCANLINE	O dispositivo pode desenhar uma única linha de pixels.
PC_WIDE	O dispositivo pode desenhar bordas largas.
PC_STYLED	O dispositivo pode desenhar bordas estilizadas.
PC_WIDESTYLED	O dispositivo pode desenhar bordas que são largas e estilizadas.
PC_INTERIORS	O dispositivo pode desenhar interiores.

Finalmente, seus programas, com freqüência, irão requerer informações sobre as capacidades de exibição de texto de um dispositivo. Seus programas podem verificar as capacidades de texto de um dispositivo com uma chamada à *TEXTCAPS*. As chamadas à *TEXTCAPS* retornarão um ou mais dos valores que a Tabela 1426.7 detalha.

Tabela 1426.7 Os valores de retorno possíveis de uma chamada à ***TEXTCAPS***.

Bit	Significado
TC_OP_CHARACTER	Dispositivo capaz de precisão na saída do caractere.
TC_OP_STROKE	Dispositivo capaz de precisão na saída do traço.
TC_CP_STROKE	Dispositivo capaz de precisão no corte do traço.
TC_CR_90	Dispositivo capaz de rotação de caractere em 90 graus.
TC_CR_ANY	Dispositivo capaz de qualquer rotação do caractere.
TC_SF_X_YINDEP	O dispositivo pode escalar independentemente nas direções x e y.
TC_SA_DOUBLE	Dispositivo capaz de caractere dobrado para ajuste da escala
TC_SA_INTEGER	O dispositivo usa múltiplos inteiros somente para ajustar a escala do caractere.
TC_SA_CONTIN	O dispositivo usa qualquer múltiplo para ajuste exato da escala.
TC_EA_DOUBLE	O dispositivo pode desenhar caracteres de peso dobrado.
TC_IA_ABLE	O dispositivo pode italizar.
TC_UA_ABLE	O dispositivo pode sublinhar.
TC_SO_ABLE	O dispositivo pode desenhar riscado.
TC_RA_ABLE	O dispositivo pode desenhar fontes de varredura.
TC_VA_ABLE	O dispositivo pode desenhar fontes vetoriais.
TC_RESERVED	Reservado: precisa ser zero.
TC_SCROLLBLT	O dispositivo não pode usar transferência de bloco de bits para rolagem.

Para compreender melhor o processamento que a função *GetDeviceCaps* executa, considere o programa *Get_DevC.cpp*, contido no CD-ROM que acompanha este livro. O programa usa a função *CreateIC* para criar um contexto de informações para a tela. O programa *Get_DevC.cpp* usa o contexto criado para localizar o nú-

mero de bits por pixel e o número de plano de cores para a exibição. O programa *Get_DevC.cpp* usa *GetDeviceCapabilities* para fazer cada determinação. O processamento operativo é, como normal, dentro da função *WndProc*.

USANDO A FUNÇÃO GETSYSTEMMETRICS PARA ANALISAR UMA JANELA 1427

Seus programas podem manipular praticamente todas as características de uma janela, seja uma janela-mãe, uma janela-filha ou uma janela de controle. Uma das manipulações mais comuns que seus programas precisam executar é ajustar o tamanho da tela atual conforme apropriado para o tamanho da tela de um computador. Você pode obter essa informação, bem como outras informações sobre as configurações de sistema de um computador, com a função *GetSystemMetrics*. Essa função recupera várias medidas e as configurações do sistema. As medidas do sistema são as dimensões (larguras e alturas) dos elementos de exibição do Windows. Todas as dimensões que *GetSystemMetrics* lê estão em pixels. Você usará essa função dentro de seus programas, como segue:

```
int GetSystemMetrics(int nIndice);
```

O parâmetro *nIndice* especifica a medida do sistema ou a configuração que *GetSystemMetric* deve obter. Todos os valores *SM_CX* são larguras. Todos os valores *SM_CY* são alturas. A API do Windows define os valores listados na Tabela 1427.1

Tabela 1427.1 Medidas possíveis a recuperar.

Valor	Significado
SM_ARRANGE	Sinalizadores que especificam como o sistema organizou as janelas minimizadas.
SM_CLEANBOOT	Valor que especifica como o usuário iniciou o sistema. Os três valores possíveis são 0 Iniciação Normal, 1 Iniciação no Modo de Segurança e 2 Modo de Segurança com iniciação em rede. Uma iniciação no modo de segurança (também chamado *Iniciação Segura*) ignora os arquivos de iniciação do usuário.
SM_CMOUSEBUTTONS	Número de botões no mouse, ou 0 se o mouse não estiver instalado.
SM_CXBORDER	Largura, em pixels, da borda de uma janela. Isso é equivalente ao valor *SM_CXEDGE* para janelas com o aspecto de 3-D.
SM_CYBORDER	Altura, em pixels, da borda de uma janela. Isso é equivalente ao valor *SM_CYEDGE* para janelas com o aspecto de 3-D.
SM_CXCURSOR	Largura, em pixels, de um cursor Essa é a dimensão do cursor que o controlador de vídeo atual suporta. O sistema não pode criar cursores de outros tamanhos.
SM_CYCURSOR	Altura, em pixels, de um cursor. Essa é a dimensão do cursor que o controlador de vídeo atual suporta. O sistema não pode criar cursores de outros tamanhos.
SM_CXDLGFRAME	O mesmo que *SM_CXFIXEDFRAME*.
SM_CYDLGFRAME	O mesmo que *SM_CYFIXEDFRAME*.
SM_CXDOUBLECLK	Largura, em pixels, do retângulo em torno do local de um primeiro clique em uma seqüência de duplo clique. O segundo clique precisa ocorrer dentro desse retângulo para que o sistema considere os dois cliques como um duplo clique. (Os dois cliques também precisam ocorrer dentro de um tempo especificado.)

Tabela 1427.1 Medidas possíveis a recuperar. (Continuação)

Valor	Significado
SM_CYDOUBLECLK	Altura, em pixels, do retângulo em torno do local de um primeiro clique em uma seqüência de duplo clique. O segundo clique precisa ocorrer dentro desse retângulo para que o sistema considere os dois cliques como um duplo clique (Os dois cliques também precisam ocorrer dentro de um tempo especificado.)
SM_CXDRAG	Largura, em pixels, de um retângulo centralizado em um ponto de arrastar para permitir movimento limitado do ponteiro do mouse antes do início de uma operação de arrastar. Isso permite que o usuário dê um clique e libere o botão do mouse facilmente sem iniciar não intencionalmente uma operação de arrastar.
SM_CYDRAG	Altura, em pixels, de um retângulo centralizado em um ponto de arrastar que permite ao usuário mover o ponteiro do mouse antes do início de uma operação de arrastar. Esse retângulo permite que o usuário dê um clique e libere o botão do mouse facilmente sem iniciar não intencionalmente uma operação de arrastar.
SM_CXEDGE	Largura, em pixels, de uma borda em 3-D. *SM_CXEDGE* é uma correspondente de 3-D de *SM_CXBORDER.*
SM_CYEDGE	Altura, em pixels, de uma borda em 3-D. *SM_CYEDGE* é uma correspondente de 3-D de *SM_CYBORDER.*
SM_CXFIXEDFRAME	Espessura, em pixels, da moldura em torno do perímetro de uma janela que tem uma legenda mas não é ajustável *SM_CXFIXEDFRAME* é a largura da borda horizontal. O mesmo que *SM_CXDLGFRAME.*
SM_CYFIXEDFRAME	Espessura, em pixels, da moldura em torno do perímetro de uma janela que tem uma legenda mas não é ajustável. *SM_CYFIXEDFRAME* é a altura da borda horizontal. O mesmo que *SM_CYDLGFRAME*
SM_CXFRAME	O mesmo que *SM_CXSIZEFRAME.*
SM_CYFRAME	O mesmo que *SM_CYSIZEFRAME.*
SM_CXFULLSCREEN	Largura da área cliente para uma janela de tela cheia. Para obter as coordenadas da porção da tela que a bandeja não obscurece, chame a função *SystemParametersInfo* com o valor que *GetSystemMetrics* retorna.
SM_CYFULLSCREEN	Altura da área cliente para uma janela de tela cheia. Para obter as coordenadas da porção da tela que a bandeja não obscurece, chame a função *SystemParametersInfo* com o valor *SPI_GETWORKAREA.*
SM_CXHSCROLL	Largura, em pixels, do mapa de bits de seta em uma barra de rolagem horizontal.
SM_CYHSCROLL	Altura, em pixels, de uma barra de rolagem horizontal.
SM_CXHTHUMB	Largura, em pixels do quadradinho em uma barra de rolagem horizontal.
SM_CXICON	Largura padrão, em pixels, de um ícone Esse valor é tipicamente 32x32, mas pode variar dependendo do hardware de vídeo instalado A função *LoadIcon* somente pode carregar ícones dessas dimensões.
SM_CYICON	A altura padrão, em pixels, de um ícone Este valor tipicamente é 32x32, mas pode variar dependendo do hardware de vídeo instalado A função LoadIcon somente pode carregar ícones destas dimensões.
SM_CXICONSPACING	Dimensão X, em pixels, de uma célula de grade para itens no modo de exibição de ícones grandes. Cada item cabe em um retângulo desse tamanho quando organizado. Esses valores são sempre maiores ou iguais a *SM_CXICON* e *SM_CYICON.*

Tabela 1427.1 Medidas possíveis a recuperar. (*Continuação*)

Valor	Significado
SM_CYICONSPACING	Dimensão Y, em pixels, de uma célula de grade para itens no modo de exibição de ícones grandes. Cada item cabe em um retângulo desse tamanho quando organizado. Esses valores são sempre maiores ou iguais a *SM_CXICON* e *SM_CYICON*.
SM_CXMAXIMIZED	Dimensão X padrão, em pixels, de uma janela de alto nível maximizada.
SM_CYMAXIMIZED	Dimensão Y padrão, em pixels, de uma janela de alto nível maximizada
SM_CXMAXTRACK	Largura máxima padrão, em pixels, de uma janela que tem um título e bordas para o ajuste do tamanho. O usuário não pode arrastar a borda da janela para um tamanho maior que essa largura. Uma janela pode processar a mensagem *WM_GETMINIMAXINFO* para ignorar esses valores.
SM_CYMAXTRACK	Altura máxima padrão, em pixels, de uma janela que tem um título e bordas para o ajuste do tamanho. O usuário não pode arrastar a borda da janela para um tamanho maior que essa altura. Uma janela pode processar a mensagem *WM_GETMINIMAXINFO* para ignorar esses valores.
SM_CXMENUCHECK	Largura, em pixels, do mapa de bits da marca de verificação de menu padrão.
SM_CYMENUCHECK	Altura, em pixels, do mapa de bits da marca de verificação de menu padrão.
SM_CXMENUSIZE	Largura, em pixels, dos botões da barra de menu, tais como fechar filha, de múltiplos documentos (MIDI).
SM_CYMENUSIZE	Altura, em pixels, dos botões da barra de menu, tais como fechar filha, de múltiplos documentos (MIDI).
SM_CXMIN	Largura mínima, em pixels, de uma janela.
SM_CYMIN	Altura mínima, em pixels, de uma janela.
SM_CXMINIMIZED	Largura, em pixels, de uma janela minimizada.
SM_CYMINIMIZED	Altura, em pixels, de uma janela minimizada.
SM_CXMINSPACING	Dimensões, em pixels, de uma célula de grade para janelas minimizadas. Cada janela minizada cabe em um retângulo deste tamanho quando organizada. Esses valores são sempre maiores ou iguais a *SM_CXMINIMIZED* e *SM_CYMINIMIZED*.
SM_CYMINSPACING	Dimensões, em pixels, de uma célula de grade para janelas minimizadas. Cada janela minimizada cabe em um retângulo desse tamanho quando organizada. Esses valores são sempre maiores ou iguais a *SM_CXMINIMIZED* e *SM_CYMINIMIZED*.
SM_CXMINTRACK	Largura e altura mínimas, em pixels, de uma janela. O usuário não pode arrastar a moldura da janela até uma de tamanho menor do que essas dimensões. Uma janela pode processar a mensagem *WM_GETMINMAXINFO* para ignorar esses valores.
SM_CYMINTRACK	Largura e altura mínimas, em pixels, de uma janela O usuário não pode arrastar a moldura da janela até uma de tamanho menor do que essas dimensões. Uma janela pode processar a mensagem *WM GETMINMAXINFO* para ignorar esses valores.
SM_CXSCREEN	Largura, em pixels, da tela.
SM_CYSCREEN	Altura, em pixels, da tela.

Tabela 1427.1 Medidas possíveis a recuperar. (*Continuação*)

Valor	Significado
SM_CXSIZE	Largura, em pixels, de um botão da barra de título de uma janela.
SM_CYSIZE	Altura, em pixels, de um botão da barra de título de uma janela
SM_CXSIZEFRAME	Espessura, em pixels, da borda de ajuste do tamanho em volta do perímetro de uma janela que o usuário pode redefinir. *SM_C1SIZEFRAME* é a largura da borda horizontal O mesmo que *SM_CXFRAME.*
SM_CYSIZEFRAME	Espessura, em pixels, da borda de ajuste do tamanho em volta do perímetro de uma janela que o usuário pode redefinir. *SM_CXSIZEFRAME* é a largura da borda horizontal. O mesmo que *SM_CYFRAME.*
SM_CXSMICON	Dimensões recomendadas, em pixels, de um ícone pequeno. Os ícones pequenos tipicamente aparecem em títulos de janela e no modo de exibição de ícones pequenos.
SM_CYSMICON	Dimensões recomendadas, em pixels, de um ícone pequeno. Os ícones pequenos tipicamente aparecem em títulos de janela e no modo de exibição de ícones pequenos.
SM_CXSMSIZE	Dimensões, em pixels, dos pequenos botões do título.
SM_CYSMSIZE	Dimensões, em pixels, dos pequenos botões do título.
SM_CXVSCROLL	Largura, em pixels, da barra de rolagem vertical, e altura, em pixels, do mapa de bits da seta em uma barra de rolagem vertical.
SM_CYVSCROLL	Altura, em pixels, da barra de rolagem vertical, e largura, em pixels, do mapa de bits da seta em uma barra de rolagem horizontal.
SM_CYCAPTION	Altura, em pixels, da área de título normal.
SM CYKANJIWINDOW	Para as versões de conjunto de caractere de byte duplo do Windows, altura, em pixels, da janela Kanji na parte inferior da tela.
SM_CYMENU	Altura, em pixels, da barra de menu de uma única linha.
SM_CYSMCAPTION	Altura, em pixels, de um título pequeno.
SM_CYVTHUMB	Altura, em pixels, do quadradinho em uma barra de rolagem vertical.
SM_DBCSENABLED	*True* ou diferente de zero, se a versão do *USER.EXE* com o conjunto de caracteres de byte dobrado (DBCS) estiver instalada; caso contrário, *False* ou zero.
SM_DEBUG	*True* ou diferente de zero, se a versão de depuração de *USER.EXE* estiver instalada; caso contrário, *False* ou zero.
SM_MENUDROPALIGNMENT	*True*, ou diferente de zero, se menus suspensos estiverem alinhados à direita com relação ao item correspondente na barra de menu; *False* ou zero, se estiverem alinhados à esquerda.
SM_MIDEASTENABLED	*True*, se o sistema estiver instalado para os idiomas árabe/hebraico.
SM_MOUSEPRESENT	*True* ou diferente de zero se o mouse estiver instalado; *False*, ou zero, em caso contrário.
SM_MOUSEWHEELPRESENT	Sob o Windows NT somente, *True* ou diferente de zero se um mouse com bolinha estiver instalado; *False*, ou zero, em caso contrário.
SM_NETWORK	A função retornará um valor com o bit menos significativo ligado se uma rede estiver presente; caso contrário, o valor de retorno terá o bit menos significativo zerado. O Windows reserva os outros bits para uso futuro.
SM_PENWINDOWS	True, ou diferente de zero se as extensões da Microsoft Windows for Pen Computing estiverem instaladas; *False* em caso contrário.

Tabela 1427.1 Medidas possíveis a recuperar. (*Continuação*)

Valor	Significado
SM_SECURE	*True* se segurança estiver presente, *False* em caso contrário.
SM_SHOWSOUNDS	*True* ou diferente de zero se o usuário requiser que um aplicativo apresente informações visualmente em situações onde ele as apresentaria somente em forma audível; *False*, ou zero, em caso contrário.
SM_SLOWMACHINE	*True* se o computador tiver um processador lento; *False* em caso contrário.
SM_SWAPBUTTON	*True* ou diferente de zero se os botões esquerdo e direito do mouse estiverem invertidos; *False*, ou zero em caso contrário.

Se a função for bem-sucedida, o valor de retorno será a medida do sistema solicitada ou a definição da configuração. Se a função falhar, o valor de retorno será zero.

As medidas do sistema podem variar de um monitor de vídeo para outro. A definição *SM_ARRANGE* especifica como o sistema organiza as janelas minimizadas e consiste de uma posição inicial e uma direção. A posição inicial pode ser um dos valores listados na Tabela 1427.2.

Tabela 1427.2 Valores da posição inicial.

Valor	Significado
ARW_BOTTOMLEFT	Inicia no canto inferior esquerdo da tela (posição normal).
ARW_BOTTOMRIGHT	Inicia no canto inferior direito da tela. Equivalente a *ARW_STARTRIGHT*.
ARW_HIDE	Oculta as janelas minimizadas movendo-as para fora da área visível da tela.
ARW_TOPLEFT	Inicia no canto superior esquerdo da tela. Equivalente a *ARV_STARTTOP*.
ARW_TOPRIGHT	Inicia no canto superior direito da tela. Equivalente a *ARW_STARTTOP* | *SRW_STARTRIGHT*.

A direção em que o sistema organiza as janelas minimizadas pode ser um dos valores listados na Tabela 1427.3.

Tabela 1427.3 Valores de organização possíveis.

Valor	Significado
ARW_DOWN	Organiza verticalmente, de cima para baixo
ARW_LEFT	Organiza horizontalmente, da esquerda para a direita
ARW_RIGHT	Organiza horizontalmente, da direita para a esquerda
ARW_UP	Organiza verticalmente, de baixo para cima

A Dica 1428, a seguir, detalha como seus programas podem usar as medidas do sistema durante o processamento.

COMPREENDENDO OS USOS PARA GETSYSTEMMETRICS 1428

Como você aprendeu na dica anterior, seus programas podem usar a função *GetSystemMetrics* para obter informações sobre as medidas atuais do sistema em qualquer computador. À medida que seus programas se tornarem mais complexos, e que você criar aplicativos que possam rodar em diferentes computadores, torna-se importante conferir as medidas do sistema antes de gerar tela, pois ajuda a garantir que suas telas tenham a aparência apropriada tanto em monitores de 640x480 quanto em 1024x768. Por exemplo, o programa *Get_Sysm.cpp*, contido no CD-ROM que acompanha este livro, usa *GetSystemMetrics* para enumerar o status atual do tamanho de uma janela, o tamanho da tela e outras informações variadas. Quando o usuário seleciona a opção *Testar!*, o programa exibe informações dentro da janela.

1429 OBTENDO UM DISPOSITIVO DO CONTEXTO PARA UMA JANELA INTEIRA

Seus programas geralmente trabalham com o dispositivo do contexto para a área cliente de uma janela. No entanto, algumas vezes seus programas irão requerer um dispositivo do contexto para uma janela inteira. Para isso, seus programas podem usar a função *GetWindowDC*. Essa função obtém o dispositivo do contexto (DC) para a janela inteira, incluindo a barra de título, menus e barras de rolagem. Um dispositivo do contexto em janela permite a pintura em qualquer lugar em uma janela, pois a origem do dispositivo do contexto é o canto superior esquerdo da janela e não da área cliente.

GetWindowDC atribui atributos padrão para o dispositivo do contexto da janela toda vez que o lê. Como *GetWindowDC* atribui atributos padrão ao dispositivo de contexto em janela, os atributos anteriores são perdidos. Você usará a função *GetWindowDC* dentro de seus programas como mostrado no seguinte protótipo:

```
HDC GetWindowDC(HWND hJan);
```

O parâmetro *hJan* identifica a janela com um dispositivo do contexto que *GetWindowDC* deve recuperar. Se a função for bem-sucedida, o valor de retorno será o indicativo de um dispositivo do contexto para a janela especificada. Se a função falhar, o valor de retorno será *NULL*, indicando um erro ou um parâmetro *hJan* inválido.

GetWindowDC destina-se a efeitos especiais de pintura dentro da área não-cliente de uma janela. Pintar em áreas não-cliente de qualquer janela não é recomendado. Você pode usar a função *GetSystemMetrics* para obter as dimensões de várias partes da área não-cliente, tais como a barra de título, menu e barras e rolagem. Após a pintura estar completa, o programa precisará chamar a função *ReleaseDC* para liberar o dispositivo do contexto. Não liberar o dispositivo do contexto em janela tem sérios efeitos na pintura que os aplicativos requisitarem subsequentemente.

Para compreender melhor o processamento que *GetWindowDC* executa, considere o programa *GetWinDC.cpp* no CD-ROM que acompanha este livro. O programa *GetWinDC.cpp* usa a função *GetWindowDC* para recuperar o dispositivo do contexto para a janela inteira. Ele depois usa *GetSystemMetrics* para determinar os tamanhos das bordas e da barra de título. Finalmente, ele usa o dispositivo do contexto para a janela inteira para pintar um padrão na barra de título da janela. O processamento operativo ocorre dentro da função *WndProc*.

1430 LIBERANDO OS DISPOSITIVOS DO CONTEXTO

Como você aprendeu, liberar um objeto após seu programa completar seu processamento nesse objeto é uma parte importante da boa programação em C++. Tratar dos dispositivos do contexto não é exceção. A função *ReleaseDC* libera um dispositivo do contexto (DC), liberando-o para que outros aplicativos possam usá-lo. O efeito da função *ReleaseDC* depende do tipo de dispositivo do contexto. *ReleaseDC* libera somente os dispositivos do contexto comuns e de janela, não tendo efeito em dispositivos do contexto de classe ou privados. Você usará a função *ReleaseDC* dentro de seus programas, como mostrado no seguinte protótipo:

```
int ReleaseDC(HWND hJan, HDC hDC);
```

O parâmetro *hjan* identifica a janela cujo dispositivo do contexto *ReleaseDC* deverá liberar. O parâmetro *hDC* identifica o dispositivo do contexto que *ReleaseDC* deve liberar. O valor de retorno especifica se *ReleaseDC* liberou com sucesso o dispositivo do contexto. Em caso afirmativo, a função retornará 1; caso contrário, retornará 0.

O aplicativo precisa chamar a função *ReleaseDC* para cada chamada à *GetWindowDC* e para cada chamada à função *GetDC* que lê um dispositivo do contexto comum. Um aplicativo não pode usar a função *ReleaseDC* para liberar um dispositivo do contexto criado pelo programa com a função *CreateDC*. Em vez disso, o aplicativo precisa usar *DeleteDC*.

OBTENDO UM INDICATIVO DE UMA JANELA A PARTIR DO DISPOSITIVO DO CONTEXTO 1431

Seus programas podem freqüentemente executar processamento generalizado em qualquer dispositivo do contexto. No entanto, você pode querer evitar executar esse processamento no dispositivo do contexto associado com uma determinada janela. Com a função *WindowFromDC*, seus programas podem converter um dispositivo do contexto em um indicativo de janela. A função *WindowFromDC* retorna o indicativo da janela associada com o dispositivo do contexto (DC) de vídeo dado. As funções de saída que usam o dispositivo do contexto desenham na janela cujo indicativo *WindowFromDC* retorna. Seus programas usarão a função *WindowFromDC*, como segue:

```
HWND WindowFromDC(HDC hDC);
```

O parâmetro *hDC* identifica o dispositivo do contexto a partir do qual *WindowFromDC* deve recuperar um indicativo para a janela associada. Se a função for bem-sucedida, o valor de retorno será o indicativo da janela associada com o dispositivo do contexto de vídeo dado. Se a função falhar, o valor de retorno será *NULL*.

COMPREENDENDO OS MAPAS DE BITS DEPENDENTES DO DISPOSITIVO 1432

Mapas de bits (ou *bitmaps*) são blocos de dados que seus programas podem exibir diretamente em um dispositivo, tal como o monitor de vídeo. Você pode pensar nos mapas de bits como um modo de armazenar os dados de pixels diretamente da tela em uma área de memória. Pintar mapas de bits na tela é muito mais rápido do que usar as funções da interface de dispositivo gráfico, tais como *Rectangle* e *LineTo*. Os aspectos negativos dos mapas de bits são que eles consomem muita memória e espaço em disco e não permitem um bom ajuste da escala, particularmente se contiverem texto. Quando você ajusta a escala de um mapa de bits, ele perde qualidade e distorce o texto.

O Windows fornece dois tipos de mapas de bits: *mapas de bits dependentes e independentes do dispositivo*. O mapa de bits dependente do dispositivo é um formato mais antigo que é, como seu nome implica, menos flexível do que o mapa de bits independente do dispositivo. Todos os aplicativos Win32 que você escrever devem usar mapas de bits independentes do dispositivo, pois o Windows fornece a maioria das funções Win32 que lida com mapas de bits dependentes do dispositivo unicamente por questões de compatibilidade com os aplicativos Windows de 16 bits.

Você normalmente cria mapas de bits com um programa de pintura, tal como o Microsoft *Paint* ou dentro de um componente editor de mapas de bits do ambiente integrado de desenvolvimento (tal como o Borland C++ 5.02 ou o Visual C++ 5.0). Você então armazena o mapa de bits em um disco com uma extensão de nome de arquivo *BMP*. O Windows salvará os mapas de bits como mapas de bits independentes do dispositivo e converterá para um mapa de bits dependentes do dispositivo quando seu programa chamar *LoadBitmap*. Você pode preceder o nome de arquivo do mapa de bits com a palavra-chave *BITMAP*, e acrescentar mapas de bits a um arquivo de recursos, como mostrado aqui:

```
caneta BITMAP caneta.bmp
```

Para compreender melhor o processamento que seus programas executam para manipular mapas de bits, considere o programa *Pen_BMP*, contido no CD-ROM que acompanha este livro. O programa *Pen_BMP* carrega o mapa de bits de caneta, coloca-o em um dispositivo do contexto em memória e depois o pinta no dispositivo do contexto da tela. O fragmento de código do arquivo *Pen_BMP.cpp* a seguir mostra o processamento operativo:

```
HBITMAP  hBitmap;
HDC      hDC;
HDC      hMemDC;

//  Carrega o mapa de bits na memória

hBitmap = LoadBitmap(hInst, "pen");

//  Pinta o mapa de bits no MemDC e depois a tela
```

```
hDC = GetDc(hWnd);
hMemDC = CreateCompatibleDC(hDC);
SelectObject(hMemDC, hBitmap);
BitBlt(hDC, 10, 10, 60, 60, hMemDC, 0, 0, SRCCOPY);
DeleteDC(hMemDC);
ReleaseDC(hWnd, hDC);
DeleteObject(hBitmap);
```

O formato de mapa de bits dependentes do dispositivo funciona bem para copiar seções da tela na memória e colocar essas seções de volta em outros locais da tela. No entanto, quando o aplicativo precisa salvar dados em um arquivo no disco, depois exibi-lo em outro dispositivo, o formato de mapa de bits dependentes do dispositivo deixa a desejar. O formato de mapa de bits dependentes do dispositivo assume que você sempre exibirá o mapa de bits em um dispositivo similar ao que ele foi criado originalmente, e as cores serão as mesmas no segundo dispositivo e no primeiro. O resultado infeliz é que, se você exibir o mapa de bits em outro dispositivo, as cores poderão ser diferentes. O mapa de bits independentes do dispositivo evita os problemas do formato de mapa de bits dependentes do dispositivo.

1433 COMPREENDENDO OS MAPAS DE BITS INDEPENDENTES DO DISPOSITIVO

Como você aprendeu na dica anterior, o formato de mapa de bits dependentes do dispositivo tem algumas limitações. O formato de mapa de bits independentes do dispositivo supera essas limitações. A diferença mais significativa entre um mapa de bits dependentes do dispositivo e um mapa de bits independentes do dispositivo é que este último inclui uma tabela das cores que o mapa de bits usará. O cabeçalho do mapa de bits também é mais complexo no formato independente do dispositivo. Ao contrário de um mapa de bits dependentes do dispositivo, um independente do dispositivo não é um objeto gráfico; em vez disso, é um formato de dados. Um aplicativo não pode selecionar um mapa de bits independentes do dispositivo em um dispositivo do contexto. O formato do mapa de bits independentes do dispositivo consiste de três seções de dados separadas. A primeira é a estrutura *BITMAPINFOHEADER*, que a API do Windows define como mostrado aqui:

```
typedef struct tagBITMAPINFOHEADER{
    DWORD biSize;
    LONG  biWidth;
    LONG  biHeight;
    WORD  biPlanes;
    WORD  biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG  biXPelsPerMeter;
    LONG  biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

A estrutura *BITMAPINFOHEADER* contém os membros que a Tabela 1433.1 detalha.

Tabela 1433.1 Os membros da estrutura ***BITMAPINFOHEADER***.

Nome do Membro	Descrição
biSize	Especifica o número de bytes que a estrutura requer.
biWidth	Especifica a largura do mapa de bits, em pixels.
biHeight	Especifica a altura do mapa de bits, em pixels. Se *biHeight* for positivo, o mapa de bits será um mapa de bits independentes do dispositivo de baixo para a cima, e sua origem será o canto inferior esquerdo. Se *biHeight* for negativo, o mapa de bits será o mapa de bits independentes do dispositivo de cima para baixo, e sua origem será o canto superior esquerdo.

Tabela 1433.1 Os membros da estrutura ***BITMAPINFOHEADER***. (*Continuação*)

Nome do Membro	Descrição
biPlanes	Especifica o número de planos para o dispositivo-alvo Ese valor precisa ser deixado como 1.
biBitCount	Especifica o número de bits por pixel. Esse valor precisa ser 1, 4, 8, 16, 24 ou 32.
biCompression	Especifica o tipo de compressão de um mapa de bits comprimido de baixo para a cima (o Windows não comprimirá os mapas de bits independentes do dispositivo de cima para baixo). Pode ser um dos valores detalhados na Tabela 1433.2.
biSizeImage	Especifica o tamanho, em bytes, da imagem. Isso pode ser deixado como zero para os mapas de bits *BI_RGB*.
biXPelsPerMeter	Especifica a resolução horizontal, em pixels por metro, do dispositivo-alvo para o mapa de bits. Um aplicativo pode usar esse valor para selecionar um mapa de bits a partir de um grupo de recurso que se aproxima melhor das características do dispositivo atual.
biYPelsPerMeter	Especifica a resolução vertical, em pixels por metro, do dispositivo-alvo para o mapa de bits.
biClrUsed	Especifica o número de índice de cores na tabela de cores que o mapa de bits usa realmente. Se esse valor for zero, o mapa de bits usará o número máximo de cores que correspondem ao valor do membro *biBitCount* para o modo de compressão que *biCompression* especifica. Se *biClrUsed* for diferente de zero e o membro *biBitCount* for menor que 16, o membro *biClrUsed* especificará o membro real de cores que o sistema gráfico do controlador de dispositivo acessará. Se *biBitCount* for 16 ou maior, o membro *biClrUsed* especificará o tamanho da tabela de cores que o mapa de bits independentes do dispositivo usa para otimizar o desempenho das paletas de cores do Windows. Se *biBitCount* for igual a 16 ou 32, a paleta de cores ótima iniciará imediatamente após as três máscaras de *doublewords*. Se o mapa de bits for compactado (um mapa de bits no qual a matriz do mapa de bits segue imediatamente o cabeçalho *BITMAPINFO* e que um único ponteiro referencia), o membro *biClrUsed* precisará ser zero ou o tamanho real da tabela de cores.
biClrImportant	Especifica o número de índices de cores que a definição do mapa de bits independentes do dispositivo especifica como importante para exibir o mapa de bits. Se esse valor for zero, todas as cores serão importantes.

Como visto na Tabela 1433.1, você pode criar mapas de bits comprimidos de baixo para cima. Quando seu programa carrega um mapa de bits comprimido, deve conferir o valor do membro *biCompressed* para determinar o tipo de compressão. A Tabela 1433.2 mostra os valores possíveis para o membro *biCompressed*.

Tabela 1433.2 Valores possíveis para o membro ***biCompressed***.

Valor	Descrição
BI_RGB	Um formato não-comprimido.
BI_RLE8	Um formato codificado para mapas de bits com 8 bits por pixel. O formato de compressão é um formato de dois bytes que consiste de um byte contador seguido por um byte que contém um índice de cor.
BI_RLE4	Um formato codificado para os mapas de bits com 4 bits por pixel. O formato de compressão é um formato de dois bytes que consiste de um byte contador seguido por índices de cores do tamanho de duas palavras.

Tabela 1433.2 Valores possíveis para o membro ***biCompressed***. (*Continuação*)

Valor	Descrição
BI_BITFIELDS	Especifica que o mapa de bits não é comprimido e que a tabela de cores consiste de três máscaras de cores de doubleword que especificam os componentes vermelho, verde e azul, respectivamente, de cada pixel. O valor *BI_BITFIELDS* é válido para seus programas usem com mapas de bits de 16 ou 32 bits por pixel.

A estrutura *BITMAPINFO* combina a estrutura *BITMAPINFOHEADER* e uma tabela de cores para fornecer uma definição completa das dimensões e cores de um mapa de bits independentes do dispositivo. Um aplicativo deve usar as informações que o membro *biSize* contém para localizar a tabela de cores em uma estrutura *BITMAPINFO*, como mostrado aqui:

```
pCor = ((LPSTR)pBitmapInfo + (WORD)(pBitmapInfo->bmiHeader.biSize));
```

O Windows suporta formatos para comprimir mapas de bits que definem suas cores com 8 ou 4 bits por pixel. A compressão reduz o espaço que o mapa de bits requer em disco e na memória. Na Tabela 1433.2, você aprendeu que o Windows suporta três formatos de compressão.

Quando o membro biCompression for BI_RLE8, o programa criador originalmente comprimiu o mapa de bits com o formato codificado para um mapa de bits de 8 bits. O programa criador pode usar o formato codificado para comprimir nos modos absoluto ou codificado. Ambos os modos podem ocorrer em qualquer lugar no mesmo mapa de bits.

O modo codificado consiste de dois bytes. O primeiro especifica o número de pixels consecutivos que o Windows deve desenhar usando o índice de cor que o segundo byte contém. Além disso, o programa que criou o mapa de bits pode definir o primeiro bytes do par para zero para indicar um escape que indica um fim de linha, fim do mapa de bits ou um delta (uma modificação). A interpretação do escape depende do valor do segundo byte do par, o que pode ser um dos valores na Tabela 1433.3.

Tabela 1433.3 Valores possíveis para o segundo byte de um par de bytes

Valor	Significado
0	Fim da linha.
1	Fim do mapa de bits.
2	Delta. Os dois bytes que seguem o escape contêm valores não-atribuídos que indicam os deslocamentos horizontal e vertical do próximo pixel a partir da posição atual.

Por outro lado, no modo absoluto, o primeiro byte é zero e o segundo byte é um valor no intervalo 03H até FFH. O segundo byte representa o número de bytes seguintes, cada um dos quais contém o índice de cor de um único pixel. Quando o segundo byte for 2 ou menos, o escape terá o mesmo significado que no modo codificado. No modo absoluto, o programa criador precisa alinhar cada execução dentro do mapa de bits independentes do dispositivo em um limite de palavra.

Após a estrutura *BITMAPINFOHEADER*, um mapa de bits independentes do dispositivo contém a tabela de cores. A tabela de cores é um conjunto de estrutura de dados *RGBQUAD* que contém a cor RGB para cada cor que o mapa de bits usa. A estrutura *RGBQUAD* descreve uma cor que consiste de intensidades relativas de vermelho, verde e azul. Existirão tantos itens *RGBQUAD* quantas são as escolhas de cores no mapa de bits. A API do Windows constrói a estrutura *RGBQUAD* como mostrado aqui:

```
typedef struct tagRGBQUAD {
    BYTE  rgbBlue;
    BYTE  rgbGreen;
    BYTE  rgbRed;
    BYTE  rgbReserved;
} RGBQUAD;
```

A estrutura *RGBQUAD* consiste dos membros que a Tabela 1433.4 detalha.

Tabela 1433.4 Membros da estrutura ***RGBQUAD***.

Membro	Descrição
rgbBlue	Especifica a intensidade de azul na cor.
rgbGreen	Especifica a intensidade de verde na cor
rgbRed	Especifica a intensidade de vermelho na cor.
rgbReserved	O Windows reserva este membro, precisa ser zero.

O restante do arquivo de mapa de bits independentes do dispositivo contém os dados de pixel reais para o mapa de bits. Você aprenderá mais sobre a criação e exibição de formatos de mapa de bits dependentes e in dependentes do dispositivo em dicas posteriores.

CRIANDO MAPAS DE BITS 1434

Como você aprendeu, um dos três tipos de arquivos gráficos nativos na API do Windows é o arquivo de mapa de bits. Dentro de seus programas, você pode criar mapas de bits para exibir dentro de uma janela. Para criar mapas de bits, seus programas usarão a função *CreateBitmap*. A função *CreateBitmap* cria um mapa de bits com a largura, altura e formato de cores (cor de planos e bits por pixel) que você especificar. Você usará a função *CreateBitmap* dentro de seus programas, como segue:

```
HBITMAP CreateBitmap (
  int  nLarg,        // largura do mapa de bits, em pixels
  int  nAlt,         // altura do mapa de bits, em pixels
  UINT cPlanos,      // número de planos de cores usado pelo dispositivo
  UINT cBitsPorPixel, // número de bits requeridos para identificar uma cor
  CONST VOID *lpvBits // ponteiro para matriz que contém os dados de cor
);
```

Quando seus programas chamarem a função *CreateBitmap*, a função esperará os parâmetros que a Tabela 1434.1 detalha.

Tabela 1434.1 Parâmetros para a função ***CreateBitmap***.

Parâmetro	Descrição
nLarg	Especifica a largura do mapa de bits, em pixels.
nAlt	Especifica a altura do mapa de bits, em pixels.
cPlanos	Especifica o número de planos de cores que o dispositivo usa.
cBitsPorPixel	Especifica o número de bits que o dispositivo requer para identificar a cor de um único pixel.
lpvBits	Aponta para uma matriz de dados de cor que o dispositivo usa para definir as cores em um retângulo de pixels. Você precisa garantir que alinhe cada linha de varredura no retângulo em uma word (precisa preencher linhas de varredura que não se alinhem em words com zeros). Se esse parâmetro for *NULL*, o novo mapa de bits será indefinido.

Se a função *CreateBitmap* for bem-sucedida, o valor de retorno será um indicativo para um mapa de bits. Se a função falhar, o valor de retorno será *NULL*.

Após você criar um mapa de bits, poderá chamar a função *SelectObject* para selecionar o mapa de bits em um dispositivo do contexto. Embora seus programas possam usar a função *CreateBitmap* para criar mapas de bits coloridos, por razões de desempenho os aplicativos deverão usar *CreateBitmap* para criar mapas de bits monocromáticos e *CreateCompatibleBitmap* para criar mapas de bits coloridos. Quando seu programa seleciona um mapa de bits coloridos anteriormente retornado de *CreateBitmap* em um dispositivo do contexto, o Windows precisa garantir que o mapa de bits é igual ao formato do dispositivo do contexto no qual você está selecionando o mapa de bits.

Se o mapa de bits for monocromático, zeros representarão a cor de frente e uns representarão a cor de fundo para o dispositivo do contexto de destino. Se um aplicativo define os parâmetro *nLarg* e *nAlt* como 0, *CreateBitmap* retorna o indicativo de um mapa de bits monocromático de 1 por 1 pixel. Quando você não precisar mais do mapa de bits, chame a função *DeleteObject* para excluí-lo.

Para compreender melhor o processamento que a função *CreateBitmap* executa, considere o programa *Create_Bitmap.cpp*, contido no CD-ROM que acompanha este livro. O programa *Create_Bitmap cpp* cria um mapa de bits monocromático quando o usuário seleciona o item de menu *Testar!*. O programa seleciona o mapa de bits em um dispositivo do contexto em memória e desenha um retângulo e uma oval no mapa de bits. O programa depois exibe o mapa de bits resultante. O programa executa o processamento operativo dentro da função *WndProc*.

1435 EXIBINDO MAPAS DE BITS

Como você viu na dica anterior, seus programas precisam "empurrar" um mapa de bits para o dispositivo do contexto de vídeo para mostrar o mapa de bits. No programa *Create_Bitmap.cpp*, você criou o mapa de bits, depois o acrescentou ao dispositivo do contexto para exibi-lo. Tipicamente, seus programas usarão a função *BitBlt* para exibir um mapa de bits. A função *BitBlt* efetua uma transferência de bloco de bits dos dados da cor que corresponde a um retângulo de pixels a partir do dispositivo do contexto de origem que você especifica em um dispositivo do contexto de destino. Você precisa primeiro selecionar o mapa de bits em um dispositivo do contexto em memória (que você cria com *CreateCompatibleDC*). Em seguida, você chamará *BitBlt* para copiar o mapa de bits no dispositivo do contexto real. Como o procedimento de cópia de *BitBlt* real usa operações de varredura, você deve conferir o *RASTERCAPS* de um dispositivo antes de executar *BitBlt* com o dispositivo. Você usará *BitBlt* dentro de seus programas de acordo com o seguinte protótipo:

```
BOOL  BitBlt(
   HDC  hdcDest,   // indicativo para dispositivo do
                   // contexto de destino
   int nXDest,     // coordenada x do canto superior
                   // esquerdo do retângulo de destino
   int nYDest,     // coordenada y do canto superior
                   // esquerdo do retângulo de destino
   int nLarg,      // largura do retângulo de destino
   int nAlt,       // altura do retângulo de destino
   HDC hdcFnt,     // indicativo para dispositivo do
                   // contexto de origem
   int nXFnt,      // coordenada x do canto superior
                   // esquerdo do retângulo de origem
   int nYFnt,      // coordenada y do canto superior
                   // esquerdo do retângulo de origem
   DWORD dwRop     // código da operação de varredura
);
```

A função *BitBlt* aceita os parâmetros que a Tabela 1435 detalha.

Tabela 1435.1 Parâmetros para a função ***BitBlt***.

Parâmetro	Descrição
hdcDest	Identifica o dispositivo do contexto de destino.
nXDest	Especifica a coordenada lógica x do canto superior esquerdo do retângulo de destino.
nYDest	Especifica a coordenada lógica y do canto superior esquerdo do retângulo de destino.
nLarg	Especifica a largura lógica dos retângulos de origem e de destino.
nAlt	Especifica a altura lógica dos retângulos de origem e de destino.
hdcFnt	Identifica o dispositivo do contexto de origem.
nXFnt	Especifica a coordenada lógica x do canto superior esquerdo do retângulo de origem.
nYFnt	Especifica a coordenada lógica y do canto superior esquerdo do retângulo de origem.

Tabela 1435.1 Parâmetros para a função ***BitBlt***..

Parâmetro	Descrição
dwRop	Especifica uma código de operação de varredura. Esses códigos definem como a função deve combinar os dados de cor para o retângulo de origem com os dados de cor para o retângulo de destino para obter a cor final. A Tabela 1435.2 detalha alguns códigos comuns de operação de varredura.

Como você aprendeu, seus programas geralmente rasterizam os mapas de bits ao colocá-los em um dispositivo do contexto. A Tabela 1435.2 mostra alguns códigos de operação de varredura comuns que seus programas usarão ao colocar os mapas de bits em um dispositivo do contexto.

Tabela 1435.2 Operações de varredura comuns que seus programas executarão com os mapas de bits.

Valor	Descrição
BLACKNESS	Usa a cor associada com o índice zero na paleta física para preencher o retângulo de destino. (Esta cor é preta para a paleta física padrão.)
DSTINVERT	Inverte o retângulo de destino.
MERGECOPY	Usa o operador booleano *E* para mesclar as cores do retângulo de origem com o padrão especificado.
MERGEPAINT	Usa o operador booleano *OU* para mesclar as cores do retângulo de origem invertido com as cores do retângulo de destino
NOTSRCCOPY	Copia o retângulo de origem invertido no destino.
NOTSRCERASE	Usa o operador booleano *OU* para combinar as cores dos retângulos de origem e de destino e depois inverte a cor resultante.
PATCOPY	Copia o padrão especificado no mapa de bits de destino
PATINVERT	Usa o operador booleano *OU* Exclusivo para combinar as cores do padrão especificado com as cores do retângulo de destino.
PATPAINT	Usa o operador booleano *OU* para combinar as cores do padrão com as cores do retângulo de origem invertido *y*. Você pode usar o operador booleano *OU* para combinar o resultado dessa operação com as cores do retângulo de destino.
SRCAND	Usa o operador booleano *E* para combinar as cores dos retângulos de origem e de destino.
SRCCOPY	Copia o retângulo de origem diretamente para o retângulo de destino.
SRCERASE	Usa o operador booleano *E* para combinar as cores invertidas do retângulo de destino com as cores do retângulo de origem.
SRCINVERT	Usa o operador booleano *Ou* Exclusivo para combinar as cores dos retângulos de origem e de destino
SRCPAINT	Usa o operador booleano para combinar as cores dos retângulos de origem e de destino.
WHITENESS	Usa as cores associadas com o índice 1 na paleta física para preencher o retângulo de destino. (Esta cor é branca para a paleta física padrão.)

Se uma rotação ou uma alteração (uma que altere o comprimento e a orientação aparente das linhas verticais ou horizontais em um objeto) estiver em efeito no dispositivo do contexto de origem, *BitBlt* retornará um erro. Se outras transformações existirem no dispositivo do contexto de origem (e uma transformação correspondente não estiver em efeito no dispositivo do contexto de destino), a função *BitBlt* alargará, comprimirá ou rotacionará o retângulo no dispositivo do contexto de destino conforme for necessário.

Se os formatos de cores dos dispos.tivos do contexto de origem e de destino não forem iguais, a função *Bit Blt* converterá o formato de cor de origem para combinar com o formato de destino. Quando *BitBlt* estiver gravando um meta-arquivo expandido, um erro ocorrerá se o dispositivo do contexto de origem identificar um dis-

positivo do contexto de meta-arquivo expandido. BitBlt retornará um erro se os dispositivos do contexto de origem e de destino representarem diferentes dispositivos (o que significa que você sempre deverá criar o contexto de origem com a função CreateCompatibleDC).

Para compreender melhor o processamento que seus programas executarão com a função BitBlt, considere o programa Happy_Faces.cpp no CD-ROM que acompanha este livro. O programa Happy_Faces.cpp carrega um mapa de bits em memória, depois dispõe esse mapa de bits lado a lado da área cliente da janela. O programa executa o processamento operativo dentro da função WndProc.

1436 CRIANDO MAPAS DE BITS NA MEMÓRIA

Como você aprendeu, o Windows suporta mapas de bits dependentes e independentes do dispositivo. Na dica anterior, por exemplo, você criou um mapa de bits dependentes do dispositivo e exibiu o mapa de bits na tela. Para exibir mapas de bits independentes do dispositivo, seus programas executarão processamento similar. Você precisa converter cada mapa de bits independentes do dispositivo em um mapa de bits dependentes do dispositivo antes de poder exibi-lo dentro de um dispositivo do contexto. Você usará a função *CreateDIBitmap* para criar um mapa de bits dependentes do dispositivo a partir de um mapa de bits independentes do dispositivo e, opcionalmente, ligar os bits do mapa de bits. Você usará a função *CreateDIBitmap* dentro de seus programas, como segue:

```
HBITMAP CreateDIBitmap(
  HDC  hdc,                       // indicativo para o
                                  // dispositivo do contexto
  CONST BITMAPINFOHEADER *lpbmih, ponteiro para tamanho
                                  // do mapa de bits e dados de formato
  DWORD fdwInit,                   // sinalizador de inicialização
  CONST VOID *lpbInit,            // ponteiro para dados da inicialização
   CONST BITMAPINFO *lpbmi,       // ponteiro para dados de
                                  // mapa de bits de formato de cores
   UINT fuUso                     // uso de dados de cor
);
```

A função *CreateDIBitmap* aceita os parâmetros que a Tabela 1436.1 detalha.

Tabela 1436.1 Parâmetros para a função *CreateDIBitmap*.

Parâmetro	Descrição
hdc	Identifica o dispositivo do contexto.
lpbmih	Aponta para uma estrutura *BITMAPINFOHEADER*. Se *fdwInit* for o valor *CBM_INIT*, a função usará a estrutura *BITMAPINFOHEADER* para obter a largura e a altura desejadas do mapa de bits bem como outras informações. Observe que um valor positivo para a altura indica um mapa de bits independentes do dispositivo de baixo para cima, enquanto um valor negativo indica um mapa de bits independentes do dispositivo de cima para baixo. Esse cenário é compatível com a função *CreateDIBitmap*.
fdwInic	Um conjunto de bits sinalizadores que especifica como o sistema operacional inicializa os bits do mapa de bits. Seus programas podem especificar somente uma constante para *fdwInic*. O valor do parâmetro precisa ser *CBM_INIT* ou zero. Se esse sinalizador estiver ligado, o sistema operacional usará os dados apontados pelos parâmetros *lpbInic* e *lpbmi* para inicializar os bits do mapa de bits. Se esse sinalizador estiver zerado, a função não usará os dados apontados por esses parâmetros. Se *fdwInic* for zero, o sistema operacional não inicializará os bits do mapa de bits.

Tabela 1436.1 Parâmetros para a função *CreateDIBitmap*. (*Continuação*)

Parâmetro	Descrição
lpbInic	Aponta para uma matriz de bytes que contém os dados do mapa de bits inicial. O formato dos dados depende do membro *biBitCount* da estrutura *BITMAPINFO* para o qual o parâmetro *lpbmi* aponta.
lpbmi	Aponta para uma estrutura *BITMAPINFO* que descreve as dimensões e o formato de cor da matriz apontada pelo parâmetro *lpbInic*.
fuUso	Especifica se o programa que criou o mapa de bits inicializou o membro *bmiColors* da estrutura *BITMAPINFO* e, em caso afirmativo, se *bmiColors* contém valores vermelho, verde e azul (RGB) explícitos ou índices de paleta. O parâmetro *fuUso* precisa conter um dos valores detalhados na Tabela 1436.2.

Como você aprendeu, o parâmetro *fuUso* aceita um dentre dois valores constantes. A Tabela 1436.2 lista os valores constantes possíveis para o parâmetro *fuUso*.

Tabela 1436.2 Valores possíveis para o parâmetro *fuUso*.

Valor	Significado
DIB_PAL_COLORS	O arquivo de mapa de bits fornece uma tabela de cores e consiste de uma matriz de índices de 16 bits na paleta lógica do dispositivo do contexto no qual seu programa selecionará o mapa de bits.
DIB_RGB_COLORS	O arquivo de mapa de bits fornece uma tabela de cores e contém valores RGB literais.

Se a função for bem-sucedida, o valor de retorno será um indicativo para o mapa de bits. Se a função falhar, o valor de retorno será *NULL*. Quando você não precisar mais do mapa de bits, chame a função *DeleteObject* para excluí-lo.

Para compreender melhor o processamento que a função *CreateDIBitmap* executa, considere o programa *Create_DIB_Bitmap*, contido no CD-ROM que acompanha este livro. Esse programa carrega um mapa de bits independente do dispositivo, pinta-o em um dispositivo do contexto em memória e depois transfere esse dispositivo do contexto para um dispositivo do contexto em tela (a janela do programa). Como é normal, o código do programa dentro da função *WndProc* no programa *Create_OIB_Bitmap.cpp* executa o processamento operativo.

PREENCHENDO UM RETÂNGULO COM UM PADRÃO 1437

Seus programas podem usar mapas de bits para acrescentar informações gráficas nas janelas. Adicionalmente, seus programas podem usar canetas e dispositivos do contexto em memória para desenhar formas simples e complexas em uma janela. Por exemplo, seus programas podem usar a função *PatBlt* para desenhar retângulos em um dispositivo do contexto. A função *PatBlt* pinta o retângulo dado usando o pincel que está selecionado atualmente no dispositivo do contexto especificado. A função *PatBlt* usa a operação de varredura para combinar a cor do pincel e as cores da superfície. Você usará a função *PatBlt* dentro de seus programas, como segue:

```
BOOL PatBlt(
   HDC hdc,      // indicativo para o dispositivo do contexto
   int nXEsq,    // coord. x, do canto superior esquerdo
                 // do retângulo  a ser preenchido
   int nYEsq,    // coord. y, do canto superior esquerdo
                 // do retângulo  a ser preenchido
   int nLarg,    // largura do retângulo a ser preenchido
   int nAlt,     // altura do retângulo a ser preenchido
   DWORD dwRop   // código da operação de varredura
);
```

A função *PatBlt* aceita os parâmetros que a Tabela 1437.1 detalha.

Tabela 1437.1 Parâmetros para a função ***PatBlt***.

Parâmetros	Descrição
hdc	Identifica o dispositivo do contexto.
nXEsq	Especifica a coordenada x, em unidades lógicas, do canto superior esquerdo do retângulo que a função deverá preencher.
nYEsq	Especifica a coordenada y, em unidades lógicas, do canto superior esquerdo do retângulo que a função deverá preencher.
nLarg	Especifica a largura, em unidades lógicas, do retângulo.
nAlt	Especifica a altura, em unidades lógicas, do retângulo.
dwRop	Especifica o código da operação de varredura. Esse código pode ser um dos valores que a Tabela 1437.2 detalha.

Como você viu na Tabela 1437.1,, o parâmetro *dwRop* aceita vários códigos diferentes de operações de varredura. A Tabela 1437.2 lista os vários códigos que seus programas aceitarão.

Tabela 1437.2 Possíveis códigos de operações de varredura.

Valor	Significado
PATCOPY	Copia o padrão especificado no mapa de bits de destino.
PATINVERT	Usa o operador booleano *OU* para combinar as cores do padrão especificado com as cores do retângulo de destino.
DSTINVERT	Inverte o retângulo de destino.
BLACKNESS	Usa a cor associada com o índice zero na paleta física para preencher o retângulo de destino. (Essa cor é preta para a paleta física padrão.)
WHITENESS	Usa a cor associada com o índice 1 na paleta física para preencher o retângulo de destino. (Essa cor é branca para a paleta física padrão.)

Os valores do parâmetro *dwRop* para a função *PatBlt* são um subconjunto limitado de 256 códigos ternários de operações de varredura; em particular, você não pode usar um código de operação que referencie um retângulo de origem. Para uma lista completa dos códigos ternários de operação de varredura, consulte a Ajuda on-line do seu compilador. Nem todos os dispositivos aceitam a função *PatBlt*. Você deve usar a função *GetDeviceCaps* para verificar a capacidade *RC_BITBLT* de um dispositivo antes de chamar a função *PatBlt* para operar nele.

Para compreender melhor o processamento que a função *PatBlt* executa, considere o programa *Paint_Rect.cpp*, contido no CD-ROM que acompanha este livro. O programa pinta uma caixa cinza com uma borda tridimensional na janela do programa. Ele usa três pincéis padrão para pintar os retângulos que compõem a borda tridimensional da caixa. A função *WndProc* executa o processamento operativo.

1438 USANDO SETDIBITS

À medida que seus programas se tornarem mais complexos, algumas vezes você precisará alterar o esquema de cores de um mapa de bits. Como você aprendeu, os mapas de bits independentes do dispositivo mantêm informações de cores dentro do cabeçalho do mapa de bits. Você pode usar as cores que o programa criador armazenou anteriormente dentro de um cabeçalho de mapa de bits independentes do dispositivo para alterar as cores dentro de um determinado mapa de bits. A função *SetDIBits* usa os dados de cores que encontra no mapa de bits independentes do dispositivo especificado para definir os pixels em um mapa de bits. Você usará a função *SetDIBits*, como mostrado no protótipo a seguir:

```
int SetDIBits(
   HDC hdc,              // indicativo do dispositivo do contexto
   HBITMAP hbmp,         // indicativo do mapa de bits
   UINT uInicVar,        // linha de varredura inicial
   UINT cLinhasVar,      // número de linhas de varredura
   CONST VOID *lpvBits, // matriz de bits do mapa de bits
```

```
    CONST BITMAPINFO *lpbmi, // endereço da estrutura
                             // com dados do mapa de cores
    UINT fuCorUso            // tipo de índice de cores a usar
);
```

A função *SetDIBits* aceita os parâmetros listados na Tabela 1438.1

Tabela 1438.1 Os parâmetros para o parâmetro ***SetDIBits***.

Parâmetro	Descrição
hdc	Identifica um dispositivo do contexto.
hbmp	Identifica o mapa de bits que *SetDIBits* deve alterar usando os dados de cor do mapa de bits independentes do dispositivo especificado.
uInicVar	Especifica a linha de varredura inicial para os dados da cor independente do dispositivo na matriz apontada pelo parâmetro *lpvBits*.
cLinhasVar	Especifica o número de linhas de varredura que a função encontrou na matriz que contém dados de cor independentes do dispositivo.
lpvBits	Aponta para os dados de cor de mapa de bits independentes do dispositivo, que a função armazena como uma matriz de bytes. O formato dos valores do mapa de bits depende do membro *biBitCount* da estrutura *BITMAPINFO* apontada pelo parâmetro *lpbmi*.
lpbmi	Aponta para uma estrutura de dados *BITMAPINFO* que contém informações sobre o mapa de bits independentes do dispositivo.
fuCorUso	Especifica se a definição do mapa de bits inclui o membro *bmiColors* da estrutura *BITMAPINFO* e, em caso afirmativo, se bmiColors contém valores explícitos de vermelho, verde, azul (RGB) ou índices de paleta. O parâmetro *fuCorUso* precisa ser um dos valores mostrados na Tabela 1438.2.

Como você aprendeu na Tabela 1438.1, *fuCorUso* aceita vários valores constantes predefinidos. A Tabela 1438.2 detalha os valores possíveis para o parâmetro *fuCorUso*.

Tabela 1438.2 . Valores possíveis para o parâmetro *fuCorUso*.

Valor	Significado
DIB_PAL_COLORS	A tabela de cores consiste de uma matriz de índices de 16 bits na paleta lógica do dispositivo do contexto que o parâmetro *hdc* identifica.
DIB_RGB_COLORS	O mapa de bits fornece uma tabela de cores que contém valores RGB literais.

Se a função for bem-sucedida, o valor de retorno será o número de linhas de varredura que a função *SetDIBits* copiou com sucesso. Se a função falhar, o valor de retorno será zero. O Windows alcança velocidade ótima de desenho quando os bits do mapa de bits são índices na paleta do sistema.

Os aplicativos podem chamar a função *GetSystemPalette* para recuperar as cores e índices da paleta do sistema. Após o aplicativo recuperar as cores e os índices, ele poderá criar o mapa de bits independente do dispositivo. A função usará o dispositivo do contexto que o parâmetro *hdc* identifica somente se você definir a constante *DIB_PAL_COLORS* para o parâmetro *fuCorUso*; caso contrário, a função ignorará o parâmetro *hdc*. Seu programa ou outro programa não pode já ter selecionado o mapa de bits que o parâmetro *hbmp* identifica em um dispositivo do contexto quando o aplicativo chama a função *GetSystemPaletteEntries*. A origem para os mapas de bits independentes do dispositivo de baixo para cima é o canto superior esquerdo do mapa de bits; a origem para os mapas de bits independentes do dispositivo é o canto superior esquerdo do mapa de bits; a origem para os mapas de bits independentes do dispositivo de cima para baixo é o canto superior esquerdo do mapa de bits.

Para compreender melhor o processamento que a função *SetDIBits* executa, considere o programa *Change_Colors.cpp*, que pinta um mapa de bits independentes do dispositivo na área cliente da janela. Inicialmente, o programa pinta o mapa de bits de preto e branco. Quando o usuário seleciona o item de menu *Testar*, o programa altera os dados do mapa de bits para que quaisquer dois pixels pretos ao lado um do outro reflitam a com-

binação de pixel azul-preto. A função *WndProc* contém o código do programa que executa o processamento operativo do programa.

1439 USANDO SETDIBITSTODEVICE PARA EXIBIR UM MAPA DE BITS EM UM DETERMINADO DISPOSITIVO

Como você aprendeu na dica anterior, seus programas podem usar a função *SetDIBits* para definir os bits de cor de um mapa de bits de acordo com os valores que o cabeçalho de um determinado mapa de bits independentes do dispositivo contém. Freqüentemente, seus programas precisam reduzir as cores de um mapa de bits independentes do dispositivo para um determinado dispositivo. Por exemplo, o Windows precisará mapear um mapa de bits de 256 cores em um mapa de bits de 20 cores se tentar exibir esse mapa de bits em um monitor VGA. A função *SetDIBitsToDevice* desenha um mapa de bits independentes do dispositivo no dispositivo associado com o dado dispositivo do contexto. Adicionalmente, a função *SetDIBitsToDevice* usa dados de cor do mapa de bits independentes do dispositivo original para definir os pixels no mapa de bits desenhado. Você usará a função *SetDIBitsToDevice* dentro de seus programas, como mostrado no protótipo a seguir:

```
int SetDIBitsToDevice(
   HDC  hdc,      // indicativo do dispositivo do contexto
   int  XDest,    // coord. x do canto superior esquerdo
                  // do retângulo de destino
   int  YDest,    // coord. y do canto superior esquerdo
                  // do retângulo de destino
   DWORD dwLarg,  // largura do retângulo de origem
   DWORD dwAlt,   // altura do retângulo de origem
   int XFnt,      // coord. x do canto inferior esquerdo
                  // do retângulo de origem
   int YFnt,      // coord. y do canto inferior esquerdo
                  // do retângulo de origem
   UINT uVarInic,// primeira linha de varredura na matriz
   UINT cLinhasVar, // número de linhas de varredura
   CONST VOID *lpvBits, // endereço da matriz com os bits
                        // do mapa de bits
   CONST BITMAPINFO *lpbmi,// endereço da estrutura com
                           // informações de mapa de bits
   UINT fuCorUso           // índices de RGB ou da paleta
);
```

A função *SetDIBitsToDevice* aceita os parâmetros que a Tabela 1439.1 detalha.

Tabela 1439.1 Os parâmetros para a função ***SetDIBitsToDevice***.

Parâmetro	Descrição
hdc	Identifica o dispositivo do contexto.
XDest	Especifica a coordenada x, em unidades lógicas, do canto superior esquerdo do retângulo de destino.
YDest	Especifica a coordenada y, em unidades lógicas, do canto superior esquerdo do retângulo de destino.
dwLarg	Especifica a largura, em unidades lógicas, do mapa de bits independentes do dispositivo.
dwAlt	Especifica a altura, em unidades lógicas, do mapa de bits independentes do dispositivo.
XFnt	Especifica a coordenada x, em unidades lógicas, do canto inferior esquerdo do mapa de bits independentes do dispositivo.
YFnt	Especifica a coordenada y, em unidades lógicas, do canto inferior esquerdo do mapa de bits independentes do dispositivo.
uInicVar	Especifica a linha de varredura inicial no mapa de bits independentes do dispositivo

Tabela 1439.1 Os parâmetros para a função *SetDIBitsToDevice*. (*Continuação*)

Parâmetro	Descrição
cLinhasVar	Especifica o número de linhas de varredura no mapa de bits independentes do dispositivo na matriz apontada pelo parâmetro *lpvBits*.
lpvBits	Aponta para dados de cores do mapa de bits independentes do dispositivo que o programa criador do mapa de bits armazenou anteriormente como uma matriz de bytes.
lpbmi	Aponta para uma estrutura *BITMAPINFO* que contém informações sobre o mapa de bits independentes do dispositivo.
fuCorUso	Especifica se o membro *bmiColors* da estrutura *BITMAPINFO* contém valores explícitos de vermelho, verde e azul (RGB) ou índices em uma paleta. O parâmetro *fuCorUso* precisa ser um dos valores listados na Tabela 1438.2.

Como você aprendeu na Tabela 1439.1, o parâmetro *fuCorUso* aceitará um dos dois valores predefinidos. A Tabela 1438.2 lista os valores aceitáveis para o parâmetro *fuCorUso* com a função *SetDIBitsToDevice*.

A função alcançará a velocidade de desenho ótima do mapa de bits quando os bits do mapa de bits forem índices na paleta do sistema. Os aplicativos podem chamar a função *GetSystemPaletteEntries* para recuperar as cores e os índices da paleta do sistema. Após o aplicativo obter as cores e os índices, ele poderá criar o mapa de bits independentes do dispositivo.

A origem de um mapa de bits independentes do dispositivo é o canto inferior esquerdo; a origem de um mapa de bits independentes do dispositivo de cima para baixo é o canto superior esquerdo. Para reduzir a quantidade de memória necessária para ligar bits de um mapa de bits independentes do dispositivo grande na superfície de um dispositivo, o aplicativo pode chamar repetidamente *SetDIBitsToDevice* para enfaixar a saída, o que coloca uma porção diferente do mapa de bits na matriz *lpvBits* toda vez. Os valores dos parâmetros *cInicVar* e *cLinhasVar* identificam a porção do mapa de bits que a matriz *lpvBits* contém. A função *SetDIBitsToDevice* retornará um erro se um processo que estiver rodando no segundo plano o chamar enquanto uma seção de tela cheia do MS-DOS roda no primeiro plano.

Para compreender melhor o processamento que a função *SetDIBitsToDevice* executa, considere o programa *Draw_2_Boxes.cpp*, contido no CD-ROM que acompanha este livro. O programa *Draw_2_Boxes.cpp* usa um mapa de bits independentes do dispositivo e uma paleta personalizada. O programa pinta o mapa de bits independentes do dispositivo em seu tamanho normal com a função *SetDIBitsToDevice* e então usa a função *StretchDIBits* para exibir o mapa de bits em 200% de seu tamanho normal. O processamento operativo ocorre dentro das rotinas de tratamento de mensagem *WM_CREATE* e *WM_PAINT* na função *WndProc*.

COMPREENDENDO OS META-ARQUIVOS 1440

Em dicas anteriores, você aprendeu sobre os mapas de bits. Também aprendeu que o Windows suporta três tipos gráficos básicos. O segundo tipo é conhecido como *meta-arquivo*. Os meta-arquivos são chamados de função da interface de dispositivo gráfico (GDI) codificada. Quando um programa *reproduz* um meta-arquivo, o resultado é o mesmo, como se o programa tivesse usado diretamente as funções da interface de dispositivo gráfico. Você pode pensar em um meta-arquivo como, efetivamente, uma macro gráfica. Você pode armazenar meta-arquivos na memória (ou como um arquivo no disco), pode recarregar o meta-arquivo e pode reproduzi-lo dentro de qualquer número de aplicativos diferentes. Adicionalmente, os meta-arquivos são mais independentes do dispositivo que os mapas de bits, pois o computador (e o dispositivo) interpretará as funções da interface de dispositivo gráfico com base no dispositivo do contexto da saída.

O Windows 95 e o Windows NT suportam o meta-arquivo do Windows 3.x. No entanto ambos os sistemas operacionais Win32 também suportam o novo tipo *meta-arquivo expandido*, e seus programas devem usar meta-arquivos expandidos também. As principais diferenças entre um meta-arquivo do Windows 3.x e um meta-arquivo expandido é que os meta-arquivos expandidos são verdadeiramente independentes do dispositivo, e os meta-arquivos expandidos suportam as novas funções da API da GDI do Win32.

É importante compreender que a API Win32 suporta plenamente o tipo de meta-arquivo do Win16. Na verdade, a API Win32 contém dois conjuntos de funções de meta-arquivo — um conjunto para a API Win16 e um para a API Win32. Por exemplo, seus programas podem chamar a função *PlayMetaFile* para reproduzir um meta-arquivo de 16 bits. Do mesmo modo, seus programas podem chamar a função *PlayEnhMetaFile* para re-

produzir um meta-arquivo de 32 bits. Para clareza, as dicas deste livro enfocarão os meta-arquivos expandidos, não os meta-arquivos do Win16.

1441 CRIANDO E EXIBINDO META-ARQUIVOS

Você viu que, um meta-arquivo é uma série de instruções da interface de dispositivo gráfico que o programa criador armazenou anteriormente dentro de uma estrutura. Você usará as funções da interface de dispositivo gráfico em um dispositivo do contexto de meta-arquivo para criar meta-arquivos. Você usará um *dispositivo do contexto de referência* como a base para o meta-arquivo manter dimensões de figura entre os dispositivos de saída. O dispositivo de referência corresponde ao dispositivo no qual a função apareceu primeiro. Você usará a função *CreateEnhMetaFile* para criar meta-arquivos. A função *CreateEnhMetaFile* cria um dispositivo do contexto para um meta-arquivo de formato expandido. Você pode usar o dispositivo do contexto criado para armazenar uma figura independente do dispositivo. Você usará a função *CreateEnhMetaFile* dentro de seus programas, como segue:

```
HDC CreateEnhMetaFile(
    HDC hdcRef,           // indicativo para um dispositivo
                          // do contexto de referência
    LPCTSTR  lpNomearq,   // ponteiro para uma string do nome de arquivo
    CONST RECT *lpRetang, // ponteiro para um retângulo delimitador
    LPCTSTR lpDescricao   // ponteiro para uma string opcional de descrição
);
```

A função *CreateEnhMetaFile* aceita os parâmetros que a Tabela 1441 detalha.

Tabela 1441 Os parâmetros para a função ***CreateEnhMetaFile***.

Parâmetro	Descrição
hdcRef	Identifica um dispositivo de referência para o meta-arquivo expandido
lpNomearq	Aponta para o nome de arquivo para o meta-arquivo expandido que a função deve criar. Se esse parâmetro for *NULL*, o meta-arquivo expandido estará baseado na memória e seu conteúdo será perdido quando você usar a função *DeleteEnhMetaFile* para excluí-lo.
lpRetang	Aponta para uma estrutura *RECT* que especifica as dimensões (em unidades de 0.01 milímetro) da figura que o programa armazenará no meta-arquivo expandido.
lpDescricao	Aponta para uma string que especifica o nome do aplicativo que criou a figura, bem como o título dela. A string para a qual o parâmetro *lpDescricao* aponta precisa conter um caractere *NULL* entre o nome do aplicativo e o nome da figura e precisa terminar com dois caracteres *NULL* — por exemplo, "*Editor Gráfico XYZ\0Cobra Cega\0\0*", onde o *\0* representa o caractere *NULL*. Se *lpDescricao* for *NULL*, não haverá item correspondente no cabeçalho de meta-arquivo expandido.

O Windows usa o dispositivo de referência que o parâmetro *hdcRef* identifica para gravar a resolução e as unidades do dispositivo no qual a figura apareceu originalmente. Se o parâmetro *hdcRef* for *NULL*, o Windows usará o dispositivo de exibição atual para a referência.

Os membros *left* e *top* da estrutura RECT apontada pelo parâmetro *lpRect* precisa ser menor que os membros *right* e *bottom*, respectivamente. Os pontos na borda do retângulo são incluídos na figura. Se *lpRect* for *NULL*, a interface de dispositivo gráfico (GDI) calculará as dimensões do menor retângulo que envolve a figura desenhada pelo aplicativo. Seus programas deverão fornecer o parâmetro *lpRect* sempre que for possível.

Os aplicativos usam o dispositivo do contexto que a função *CreateEnhMetaFile* cria para armazenar uma figura gráfica em um meta-arquivo expandido. Seus programas podem passar o indicativo que identifica esse *dispositivo do contexto para qualquer função da interface de dispositivo gráfico.*

Após um aplicativo armazenar uma figura em um meta-arquivo expandido, ele pode chamar a função *PlayEnhMetaFile* para exibir a figura em qualquer dispositivo de saída. Ao exibir a figura, o Windows usa o retângulo apontado pelo parâmetro *lpRetang* e os dados de resolução do dispositivo de referência para posicionar e ajustar a escala da figura. O dispositivo do contexto que a função *PlayEnhMetaFile* retorna contém os mesmos atributos padrão associados com qualquer novo dispositivo do contexto. Os aplicativos precisam usar a função

GetWinMetaFileBits para converter um meta-arquivo expandido para o formato de meta-arquivo antigo do Windows. O nome de arquivo para um meta-arquivo expandido deve usar a extensão *EMF*

Para compreender melhor o processamento que a função *CreateEnhMetaFile* executa, considere o programa *Crosshatch_Box.cpp*, contido no CD-ROM que acompanha este livro. Esse programa cria um meta-arquivo expandido de um retângulo preenchido com um pincel hachurado. Em seguida, o programa reproduz o meta-arquivo dentro da rotina de processamento de mensagem *WM_PAINT*. O programa ajusta a escala do meta-arquivo com base na área cliente da janela — o que significa que o tamanho do retângulo mudará à medida que você alterar o tamanho da janela. As rotinas de tratamento de mensagem *WM_CREATE* e *WM_PAINT* executam o processamento operativo do programa.

ENUMERANDO OS META-ARQUIVOS EXPANDIDOS 1442

Seus programas podem usar meta arquivos para reutilizar instruções armazenadas para uma interface de dispositivo gráfico. Na verdade, você pode armazenar múltiplos conjuntos de instruções dentro de um determinado meta-arquivo. Cada conjunto de instruções é conhecido como um registro dentro do meta-arquivo. Posteriormente, quando você acessar o meta-arquivo, poderá querer acessar um conjunto específico de instruções, ou até mesmo determinar quais instruções estão dentro do meta-arquivo. Para fazer isso, seus programas podem usar a função *EnumEnhMetaFile* para listar todos os registros dentro de um meta-arquivo. A função *EnumEnhMetaFile* lê cada registro e o passa para a função de callback que você especificar para enumerar os registros dentro de um meta-arquivo no formato expandido. A função de callback processa cada registro conforme necessário. A enumeração continua até que a função de callback fornecida pelo aplicativo processe o último registro ou até que essa função retorne zero. Você usará a função *EnumEnhMetaFile* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL EnumEnhMetaFile (
   HDC hdc,                 // indicativo para o dispositivo
                            // do contexto
   HENHMETAFILE hmae,       // indicativo para meta-arquivo
                            //  expandido
   ENHMFENUMPROC  lpFuncMetaExp,  // ponteiro para função
                                  // de callback
   LPVOID lpDados,          // ponteiro para dados da função
                            // de callback
   CONST RECT *lpRetang     // ponteiro para retângulo
);                          // delimitador
```

A função *EnumEnhMetaFile* aceita os parâmetros que a Tabela 1442.1 detalha.

Tabela 1442.1 Os parâmetros para a função ***EnumEnhMetaFile***.

Parâmetro	Descrição
hdc	Identifica um dispositivo do contexto. Seus programas precisam passar esse indicativo para a função de callback.
hmae	Identifica o meta-arquivo expandido.
lpFuncMetaExp	Aponta para a função de callback fornecida pelo aplicativo.
lpDados	Aponta para dados opcionais da função de callback.
lpRetang	Aponta para uma estrutura *RECT* que especifica as coordenadas dos cantos superior esquerdo e inferior direito. A estrutura *RECT* especifica as dimensões desse retângulo em unidades lógicas.

Como você sabe, uma função de enumeração da API usa uma função de callback para processar as informações que a função de enumeração retorna. A função *EnumEnhMetaFile* usa uma função de callback com o formato generalizado da função *EnhMetaFileProc*. A função *EnhMetaFileProc* processa registros de meta-arquivo de formato expandido. Seus programas usarão a função *EnhMetaFileProc*, como mostra o protótipo a seguir:

```
int CALLBACK EnhMetaFileProc(
   HDC hDC,                        // indicativo para o
                                   // dispositivo do contexto
   HANDLETABLE FAR *lpHTabela,     // ponteiro para tabela de
                                   // indicativos de meta-arquivo
   ENHMETARECORD FAR *lpEMPR,      // ponteiro para registro
                                   // de meta-arquivo
   int nObj,                       // número de objetos
   LPARAM lpDados                  // ponteiro para dados opcionais
);
```

A função *EnhMetaFileProc* aceita os parâmetros detalhados na Tabela 1442.2.

Tabela 1442.2 Os parâmetros que a função ***EnhMetaFileProc*** aceita.

Parâmetros	Descrição
hDC	Identifica o dispositivo do contexto que seu programa passou para *EnumEnhMetaFile*
lpHTabela	Aponta para uma tabela de indicativos associados com os objetos gráficos (canetas, pincéis etc.) no meta-arquivo. O primeiro item contém o indicativo do meta-arquivo expandido.
lpEMFR	Aponta para um dos registros no meta-arquivo. Seus programas não deverão modificar esse registro. (Se a modificação for necessária, seus programas deverão efetuá-la em uma cópia do registro.)
nObj	Especifica o número de objetos com indicativos associados na tabela de indicativos.
lpDados	Aponta para quaisquer dados fornecidos pelo aplicativo.

Um aplicativo precisa passar seu endereço para a função *EnumEnhMetaFile* para registrar a função de callback. Como com as outras funções de callback que você aprendeu em dicas anteriores, *EnhMetaFileProc* é um marcador de lugar para o nome da função fornecida pelo aplicativo.

A figura inclui os pontos na borda do retângulo apontado pelo parâmetro *lpRetang*. Se o parâmetro *hDC* for *NULL*, o Windows ignorará *lpRetang*. Se a função de callback chama a função *PlayEnhMetaFileRecord*, *hDC* precisa identificar um dispositivo do contexto válido. O Windows usa a transformação e o modo de mapeamento do dispositivo do contexto para transformar a figura que a função *PlayEnhMetaFileRecord* exibe. Você pode usar a função *EnumEnhMetaFile* para incorporar um meta-arquivo expandido dentro de outro.

Para compreender melhor o processamento que a função *EnumEnhMetaFile*, considere o programa *Draw_Shapes.cpp*, contido no CD-ROM que acompanha este livro. Esse programa carrega um meta-arquivo expandido e o exibe na área cliente da janela. Quando o usuário seleciona a opção *Testar!*, o programa novamente reproduz o meta-arquivo, dessa vez usando a função *EnumEnhMetaFile*. A função de callback intercepta o registro do meta-arquivo *EMR_CREATEBRUSHINDIRECT* e o substitui com um pincel cinza claro. O processamento operativo ocorre dentro dos procedimentos *WndProc* e *PaintMetaFile*. Quando você compilar e executar o programa *Draw_Shapes.cpp*, ele primeiro desenhará as formas com hachuras, depois redesenhará com cinza sólido.

1443 USANDO A FUNÇÃO GETWINMETAFILEBITS

Seus programas Win32 devem usar a estrutura de meta-arquivo expandido. No entanto, à medida que você programar para múltiplas plataformas, algumas vezes seus programas precisarão converter um meta arquivo expandido para um meta arquivo no estilo do Windows 3.x. A função *GetWinMetaFileBits* converte registros de meta-arquivo de formato expandido em registros de meta-arquivo no formato do Windows, e armazena os registros convertidos no buffer especificado. Você usará a função *GetWinMetaFileBits* dentro de seu programa, como mostra o protótipo a seguir:

```
UINT GetWinMetaFileBits(
   HENHMETAFILE hmae,      // indicativo do meta-arquivo expandido
   UINT cbBuffer,          // tamanho do buffer
```

```
    LPBYTE lpbBuffer,       // ponteiro para buffer
    INT fnModoMapa,         // modo de mapeamento
    HDC hdcRef              // indicativo de dispositivo do
                            // contexto de referência
);
```

A função *GetWinMetaFileBits* aceita os parâmetros mostrados na Tabela 1443.

Tabela 1443 Os parâmetros para a função ***GetWinMetaFileBits***.

Parâmetro	Descrição
hmae	Identifica o meta-arquivo expandido.
cbBuffer	Especifica o tamanho, em bytes, do buffer no qual a função *GetWinMetaFileBits* deverá copiar os registros convertidos.
lpbBuffer	Aponta para o buffer no qual a função *GetWinMetaFileBits* deverá copiar os registros convertidos. Se *lpbBuffer* for *NULL*, *GetWinMetaFileBits* retornará o número de bytes que a função requer para armazenar os registros do meta-arquivo convertido.
fnModoMapa	Especifica o modo de mapeamento que seus programas devem usar com o meta-arquivo convertido.
hdcRef	Identifica o dispositivo do contexto de referência.

Se a função *GetWinMetaFileBits* for bem-sucedida e o ponteiro do buffer for *NULL*, o valor de retorno será o número de bytes que a função requer para armazenar os registros convertidos. Se a função for bem-sucedida e o ponteiro do buffer for um ponteiro válido, o valor de retorno será o tamanho dos dados do meta-arquivo em bytes. Se a função falhar, o valor de retorno será 0.

A função *GetWinMetaFileBits* converte um meta-arquivo expandido em um meta-arquivo no formato do Windows para que um aplicativo que reconhece o formato antigo possa exibir o meta-arquivo. O Windows usa o dispositivo do contexto de referência para determinar a resolução do meta-arquivo convertido. A função *GetWinMetaFileBits* não invalida o indicativo do meta-arquivo expandido. Um aplicativo deverá chamar a função *DeleteEnhMetaFile* para liberar o indicativo quando o aplicativo não precisar mais do indicativo do meta-arquivo.

Devido às limitações do método no formato do Windows, algumas informações podem ser perdidas no conteúdo do meta-arquivo obtido. Por exemplo, a função pode converter uma chamada original à função *PolyBezier* no meta-arquivo expandido em uma chamada à função *Polyline* no meta-arquivo de formato do Windows, pois não há uma função *PolyBezier* equivalente no formato do Windows.

Os aplicativos Windows 3.x definem a origem e a extensão do viewport de uma figura que o meta-arquivo no formato do Windows contém. Como resultado, os registros no formato do Windows que *GetWinMetaFileBits* cria não contêm as funções *SetViewportOrgEx* e *SetViewportExtEx*. No entanto, *GetWinMetaFileBits* cria registros no formato do Windows para as funções *SetWindowExtEx* e *SetMapMode*. Para criar um meta-arquivo de tamanho ajustável, no formato do Windows, especifique *MM_ANISOTROPIC* como o parâmetro *fnModoMapa*. O Windows sempre mapeia o canto superior esquerdo da figura no meta-arquivo com a origem do dispositivo de referência.

Para compreender melhor o processamento que a função *GetWinMetaFileBits* executa, considere o programa *Convert_EMF_WMF.cpp*, contido no CD-ROM que acompanha este livro. Esse programa converte um determinado meta-arquivo expandido (neste caso, *sample.emf*) em um meta-arquivo padrão do Windows 3.x. Em seguida, o programa salva o meta-arquivo convertido como *sample.wmf*. O processamento operativo do programa ocorre dentro da rotina de processamento da mensagem *WM_COMMAND*.

COMPREENDENDO OS ÍCONES 1444

O Windows suporta três tipos básicos de arquivos gráficos. Como você também aprendeu, seus programas podem usar o mapa de bits e o meta-arquivo para gerenciar gráficos grandes ou pequenos dentro da área cliente de uma janela e mesmo na própria janela. Os *ícones* são, na verdade, uma subclasse dos mapas de bits. No entanto, o conjunto de atividades em que você usará ícones para efetuar é limitado o suficiente para que o Windows trate os ícones como um tipo separado de arquivo gráfico.

Os ícones são pequenos mapas de bits que o Windows usa como representações visuais de objetos, tais como aplicativos, arquivos e diretórios. No Windows 95, você verá ícones em todo aspecto da interface do usuário. Em versões anteriores do Windows e no Windows NT 3.51, você verá ícones principalmente no Gerenciador de Programas.

Um típico aplicativo do Windows 95 e do Windows NT terá pelo menos dois ícones: um grande (32x32) e um pequeno (16x16). O Windows exibirá o ícone pequeno no canto superior esquerdo da janela do aplicativo quando o aplicativo estiver minimizado. O Windows usa o ícone grande para o ícone da área de trabalho do programa e nas exibições de ícones grandes.

Tipicamente, você criará ícones com o *Image Editor* do SDK do Windows 95 ou com outro editor, tal como o editor de ícones do *Visual C++*. Você então usará o comando *ICON* para acrescentar os ícones que criar no arquivo de script de recursos de um aplicativo. Um exemplo típico do uso de ícones é o registro da classe de janela principal. Como você viu nos programas nas 187 dicas anteriores, os programas registram um ícone ao registrarem a classe da janela com uma chamada à *RegisterClassEx*, como mostrado aqui:

```
int APIENTRY WinMain(HINSTANCE hCopia,
                     HINSTANCE hCopiaAnterior,
                     LPTSTR lpLinhaCmd, int nCmdMostrar)
{
   MSG      msg;
   HWND     hJan;
   WNDCLASS cj;

   // Registra a classe principal da janela do aplicativo
   // ---------------------------------------------------
   cj.style         = CS_HREDRAW | CS_VREDRAW;
   cj.lpfnWndProc   = (WNDPROC)ProcJan;
   cj.cbClsExtra    = 0;
   cj.cbWndExtra    = 0;
   cj.hInstance     = hCopia;
   cj.hIcon         = LoadIcon(hCopia, lpszNomeAplic);
   cj.hCursor       = LoadCursor(NULL, IDC_ARROW);
   cj.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
   cj.lpszMenuName  = lpszNomeAplic;
   cj.lpszClassName = lpszNomeAplic;

   if (!RegisterWin95(&cj))
         return( FALSE);
   // Mais código aqui
```

Como você pode ver, o fragmento do código anterior registra um ícone com o nome que a string *lpszNomeAplic* contém como o ícone da janela principal. Em outras dicas, você aprenderá mais sobre como anexar um ícone em um programa.

1445 CRIANDO ÍCONES

Exatamente como mapas de bits e meta-arquivos, o Windows lhe permite criar e modificar ícones em tempo de execução. Seus programas geralmente usarão a função *CreateIcon* para criar um ícone em tempo de execução. A função *CreateIcon* permite que seus programas criem ícones a partir de matrizes binárias, dados de mapa de bits e mapa de bits independentes do dispositivo. A função *CreateIcon* cria um ícone que tem o tamanho, cores e padrões de bits especificados. Você usará *CreateIcon* dentro de seus programas, como mostrado no protótipo a seguir:

```
HICON CreateIcon(
   HINSTANCE  hCopia,    // indicativo da ocorrência do aplicativo
   int nLarg,            // largura do ícone
   int nAlt,             // altura do ícone
   BYTE cPlanos,         // número de planos na máscara de
                         // bit OU Exclusivo
   BYTE cBitsPixel,      // número de bits por pixel na
                         //  máscara de bits OU Exclusivo
```

```
    CONST BYTE *lpbEbits, // ponteiro para matriz de
                          // máscara de bits E
    CONST BYTE *lpbXORbits // ponteiro para matriz de
                           // máscara de bits Ou Exclusivo
    );
```

A função *CreateIcon* aceita os parâmetros detalhados na Tabela 1445.1.

Tabela 1445.1 Os parâmetros que a função ***CreateIcon*** aceita.

Parâmetro	Descrição
hCopia	Identifica a ocorrência do módulo que está criando o ícone.
nLarg	Especifica a largura, em pixels, do ícone.
nAlt	Especifica a altura, em pixels, do ícone.
cPlanos	Especifica o número de planos na máscara de bits *OU Exclusivo* do ícone.
cBitsPixel	Especifica o número de bits por pixel na máscara de bit *OU Exclusivo* do ícone.
lpbEbits	Aponta para a matriz de bytes que contém os valores de bits para a máscara de bits *E* ícone Essa máscara de bits descreve um mapa de bits monocromático
lpbXORbits	Aponta para a matriz de bytes que contém os valores de bits para a máscara de bits *OU Exclusivo* do ícone Essa máscara de bits descreve um mapa de bits monocromático ou colorido dependente do dispositivo.

Os parâmetros *nLarg* e *nAlt* precisam especificar uma largura e uma altura que o controlador de vídeo atual suporte, pois o sistema não pode criar ícones de outros tamanhos. Para determinar a largura e a altura que o controlador de vídeo suporta, use a função *GetSystemMetrics*, especificando o valor *SM_CXICON* ou *SM_CYICON*.

A função *CreateIcon* cria o ícone a partir de dois mapas de bits (que a função usa como máscaras de bits) e máscaras de bits *E*, e a máscara de bit *OU Exclusivo*. A máscara de bit *E* é sempre um mapa de bits monocromático, com um bit por pixel. *CreateIcon* aplica a tabela da verdade na Tabela 1445.2 para as máscaras de bits *E* e *OU Exclusivo*.

Tabela 1445.2 A tabela da verdade que ***CreateIcon*** aplica para as máscaras de bit ***E*** e ***OU Exclusivo***.

Máscara de Bit E	Máscara de Bit OU Exclusivo	Exibe
0	0	Preto
0	1	Branco
1	0	Tela
1	1	Vídeo reverso

Para compreender melhor o processamento que *CreateIcon* executa, considere o programa *Create_Icon.cpp*, contido no CD-ROM que acompanha este livro. Esse programa especifica diretamente os valores de bit das máscaras *E* e *OU Exclusivo* do ícone para criar um ícone monocromático. Quando a rotina *WndProc* recebe a mensagem *WM_CREATE*, o programa cria o ícone; quando a rotina *WndProc* recebe a mensagem *WM_PAINT*, o programa pinta o ícone no vídeo.

CRIANDO ÍCONES A PARTIR DE UM RECURSO 1446

Como você aprendeu, seus programas podem criar ícones de vários modos diferentes. No entanto, seus programas geralmente não criarão dois mapas de bits e duas máscaras de bit na memória, como fez o programa *Create_Icon cpp* da dica anterior. Exatamente como com as tabelas de string e outras informações reutilizáveis, seus programas podem carregar os bits componentes de um ícone a partir de dentro de um arquivo de recurso e converter os bits para um ícone real. Para efetuar esse processamento, seus programas usarão a função *CreateIconFromResource*. Essa função cria um ícone ou um cursor a partir de bits de recursos que descrevem o ícone. Você usará a função *CreateIconFromResource*, como segue:

```
HICON CreateIconFromResource(
   PBYTE  presbits,  // ponteiro para ícone ou bits de cursor
   DWORD  dwTamRec,  // número de bytes em buffer de bit
   BOOL   fIcone,    // sinalizador de ícone ou de cursor
   DWORD  dwVer      // versão de formato do Windows
);
```

A função *CreateIconFromResource* aceita os parâmetros detalhados na Tabela 1446.1.

Tabela 1446.1 Parâmetros que a função ***CreateIconFromResource*** aceita.

Parâmetro	Descrição
presbits	Aponta para um buffer que contém os bits de recurso de ícone ou de cursor. As chamadas à *LookupIconIDFromDirectory* (no Windows 95 você também pode chamar *LookupIconIDFromDirectoryEx*) e *LoadResource* tipicamente carregam esses bits.
dwTamRec	Especifica o tamanho, em bytes, do conjunto de bits apontados pelo parâmetro *presbits*.
fIcone	Especifica se a função deve criar um ícone ou um cursor. Se esse parâmetro for *True*, a função criará um ícone. Se for *False*, a função criará um cursor.
dwVer	Especifica o número de versão do formato do ícone ou do cursor para os bits de recursos apontados pelo parâmetro *presbits*.

O parâmetro *dwVer* pode ser um dos valores mostrados na Tabela 1446.2

Tabela 1446.2 Valores para o parâmetro ***dwVer***.

Formato	dwVer
Windows 2.x	0x00020000
Windows 3.x	0x00030000

Todos os aplicativos baseados em Win32 da Microsoft usam o formato do Windows 3.x para ícones e cursores. As funções *CreateIconFromResource*, *CreateIconIndirect*, *GetIconInfo* e *LookupIconFromDirectory* (e no Windows 95, *CreateIconFromResourceEx* e *LookupIconFromDirectoryEx*) permitem que os aplicativos de interface e navegadores de ícones examinem e usem os recursos de todo o sistema.

Para compreender melhor o processamento que a função *CreateIconFromResource*, considere o programa *Display_Res_Icon.cpp*, contido no CD-ROM que acompanha este livro. Esse programa usa as várias funções de gerenciamento de recursos para localizar um ícone dentro do arquivo de recursos, encontrar sua localização no disco e carregar o ícone na memória. O programa então exibe o ícone dentro da área cliente da janela. O processamento operativo ocorre após o usuário selecionar a opção *Testar!*, e o código que executa o processamento está dentro da função *WndProc*.

1447 USANDO A FUNÇÃO CREATEICONINDIRECT

Seus programas podem criar ícones a partir de mapas de bits ou a partir de um identificador de recurso. Seus programas também podem criar ícones a partir de um valor de estrutura. Dentro de seus programas, você usará a função *CreateIconIndirect* para criar ícones a partir de componentes que você não define dentro do programa ou de um arquivo de recursos. A função *CreateIconIndirect* cria um ícone ou um cursor a partir de uma estrutura *ICONINFO*. Você usará a função *CreateIconIndirect* dentro de seus programas, como segue:

```
HICON CreateIconIndirect(PICONINFO piconinfo);
```

O ponteiro *piconinfo* aponta para uma estrutura *ICONINFO* que a função usa para criar ou o ícone ou o cursor. Se a função for bem-sucedida, o valor de retorno será o indicativo para o ícone ou o cursor que a função criou. O sistema copia os mapas de bits na estrutura *ICONINFO* antes de criar o ícone ou o cursor. O aplicativo precisa continuar a gerenciar os mapas de bits originais e excluí-los quando você não precisar mais deles. A estrutura *ICONINFO* contém informações sobre um ícone ou um cursor. A API do Windows define a estrutura *ICONINFO* como mostrado aqui.

```
typedef struct _ICONINFO {
   BOOL  fIcon;
   DWORD xHotspot;
   DWORD yHotspot;
   HBITMAP hbmMask;
   HBITMAP hbmColor;
} ICONINFO;
```

A Tabela 1447 detalha os membros da estrutura *ICONINFO*.

Tabela 1447 Os membros da estrutura ***ICONINFO***.

Membro	Descrição
fIcon	Especifica se esta estrutura define um ícone ou um cursor. O valor *True* especifica um ícone; *False* especifica um cursor.
xHotspot	Especifica a coordenada *x* do ponto de ativação do cursor. Se essa estrutura definir um ícone, o ponto de ativação sempre estará no centro do ícone, e as funções que usarem a estrutura *ICONINFO* ignorarão o membro *xHotspot*.
yHotspot	Especifica a coordenada *y* do ponto de ativação do cursor. Se essa estrutura definir um ícone, o ponto de ativação sempre estará no centro do ícone, e as funções que usarem a estrutura *ICONINFO* ignorarão o membro *xHotspot*.
hbmMask	Especifica o mapa de bits da máscara do bits do ícone. Se essa estrutura definir um ícone branco e preto, essa máscara de bits será formatada para que a metade superior seja a máscara de bits *E* do ícone, e a metade inferior seja a máscara de bits *OU Exclusivo* do ícone. Sob essa condição, a altura deve ser um múltiplo par das duas. Se essa estrutura definir um ícone de cor, essa máscara somente definirá a máscara de bits *E* do ícone.
hbmColor	Identifica o mapa de bits colorido do ícone. Esse membro poderá ser opcional se a estrutura definir um ícone branco e preto. *CreateIconIndirect* aplica a máscara de bits *E* de *hbmMask* com o sinalizador *SRCAND* no destino. Subseqüentemente, *CreateIconIndirect* usa o sinalizador *SRCINVERT* para aplicar o mapa de bits colorido (usando *Ou Exclusivo*) ao destino.

Em resumo, a estrutura *ICONINFO* define o mapa de bits monocromático e o mapa de bits colorido, e a função *CreateIconIndirect* combina os mapas de bits com base nos valores que a estrutura *ICONINFO* contém. Para compreender melhor o processamento que a função *CreateIconIndirect* executa, considere o programa *Two_icons.cpp*, contido no CD-ROM que acompanha este livro. O programa combina dois mapas de bits em um terceiro mapa de bits de ícone. Como é comum, a rotina de tratamento de mensagem dentro de *WM_COMMAND* dentro da função *WndProc* trata o processamento principal do programa.

USANDO A FUNÇÃO LOADICON 1448

Em dicas anteriores, você aprendeu como seus programas podem usar vários métodos diferentes para criar ícones em tempo de execução. No entanto, como você viu em outros programas, seus programas mais comumente usarão a função *LoadIcon* para carregar um ícone em um programa a partir de dentro do arquivo de recurso do programa. Como você aprendeu, a função *LoadIcon* fornece aos seus programas um modo fácil e eficiente de carregar ícones criados anteriormente no programa. A função *LoadIcon* carrega o recurso de ícone especificado a partir do arquivo executável (.EXE) associado com a ocorrência de um aplicativo. Você usará a função *LoadIcon* dentro de seus programas, como mostra o seguinte protótipo:

```
HICON LoadIcon(HINSTANCE hCopia, LPCTSTR lpNomeIcone);
```

O parâmetro *hCopia* identifica uma ocorrência do módulo cujo arquivo executável contém o ícone que *LoadIcon* deverá carregar. O parâmetro *hCopia* precisará ser *NULL* quando *LoadIcon* carregar um ícone padrão. O parâmetro *lpNomeIcone* apontará para uma string terminada por *NULL* que contém o nome do recurso de ícone que *LoadIcon* deverá carregar. Alternativamente, o parâmetro *lpNomeIcone* pode conter o identificador de

recurso na palavra menos significativa, e zero na palavra mais significativa. Use a macro *MAKEINTRESOURCE* para criar um valor identificador de recurso. Para usar um dos ícones predefinidos do Windows, defina o parâmetro *hCopia* como *NULL*, e o parâmetro *lpNomeIcone* como um dos valores que a Tabela 1448 detalha.

Tabela 1448 Os ícones predefinidos do Windows.

Valor	Descrição
IDI_APPLICATION	Ícone padrão do aplicativo.
IDI_ASTERISK	Asterisco (usado em mensagens informativas).
IDI_EXCLAMATION	Ponto de exclamação (usado em mensagens de advertência).
IDI_HAND	Ícone na forma de uma mão (usado em mensagens sérias de advertência).
IDI_QUESTION	Ponto de interrogação (usado em mensagens com perguntas).
IDI_WINLOGO	Logotipo do Windows.

LoadIcon carrega o recurso de ícone somente se o programa não carregou anteriormente o recurso do ícone; caso contrário, *LoadIcon* lê um indicativo ao recurso existente. A função pesquisa o recurso de ícone para o ícone mais apropriado para a tela atual. O recurso do ícone pode ser um mapa de bits colorido ou monocromático. *LoadIcon* somente pode carregar um ícone cujo tamanho segue os valores de medidas do sistema *SM_CXICON* e *SM_CYICON*. Use a função *LoadImage* para carregar ícones de outros tamanhos.

1449 USANDO LOADIMAGE PARA CARREGAR MÚLTIPLOS TIPOS GRÁFICOS

Seus programas podem usar a função *LoadIcon* para carregar um ícone a partir do arquivo de recurso do programa. Seus programas também podem usar as funções *LoadBitmap* e *LoadCursor* para carregar mapas de bits e cursores, respectivamente, a partir do arquivo de recursos do programa. Alternativamente, seus programas podem usar a função *LoadImage*, que carrega um ícone, um cursor ou um mapa de bits. Você usará a função *LoadImage* dentro de seus programas, como mostrado no protótipo a seguir:

```
HANDLE LoadImage(
    HIMAGE hCop,          // indicativo da ocorrência que contém a imagem
    LPCTSTR lpszNome,     // nome do identificador da imagem
    UINT uTipo,           // tipo da imagem
    int cxDesejado,       // largura desejada
    int cyDesejado,       // altura desejada
    UINT fuCarga          // sinalizador de carga
);
```

A função *LoadImage* aceita os parâmetros que a Tabela 1449 detalha.

Tabela 1449.1 Parâmetros para a função ***LoadImage***.

Parâmetro	Descrição
hCop	Identifica uma ocorrência do módulo que contém a imagem que *LoadImage* deverá carregar. Defina este parâmetro como zero para carregar uma imagem personalizada (*OEM*).
lpszNome	Identifica a imagem que *LoadImage* deve carregar. Se o parâmetro *hCop* não for *NULL* e o parâmetro *fuCarga* não incluir *LR_LOADFROMFILE*, *lpszNome* será um ponteiro para uma string terminada por *NULL* que contém o nome do recurso de imagem no módulo *hCop*. Por outro lado, se *hCop* for *NULL* e você não especificar a constante *LR_LOADFROMFILE*, a palavra menos significativa desse parâmetro precisará ser o identificador da imagem OEM que *LoadImage* deverá carregar. O arquivo *winuser.h* define os identificadores de imagem OEM, que têm os prefixos listados na Tabela 1449.2

Tabela 1449.1 Parâmetros para a função ***LoadImage***. (*Continuação*)

Parâmetro	Descrição
	Sob o Windows 95, se o parâmetro *fuCarga* incluir o valor *LR_LOADFROMFILE*, *lpszNome* será o nome do arquivo que contém a imagem. O Windows NT não suporta *LR_LOADFROMFILE*.
uTipo	Especifica o tipo da imagem que *LoadImage* deverá carregar. Esse parâmetro pode ser um dos valores listados na Tabela 1449.3
cxDesejado	Especifica a largura, em pixels, do ícone ou do cursor. Se esse parâmetro for zero e o parâmetro *fuCarga* for *LR_DEFAULTSIZE*, a função usará o valor da medida do sistema *SM_CXICON* ou *SM_CXCURSOR* para definir a largura. Se esse parâmetro for zero e a chamada não usar *LR_DEFAULTSIZE*, a função usará a largura do recurso atual.
cyDesejado	Especifica a altura, em pixels, do ícone ou do cursor Se esse parâmetro for zero e o parâmetro *fuCarga* for *FR_DEFAULTSIZE*, a função usará o valor da métrica do sistema *SM_CYICON* ou *SM_CXCURSOR* para definir a altura. Se o parâmetro for zero e a chamada não usar *LR_DEFAULTSIZE*, a função usará a altura do recurso atual.
fuCarga	Determina como a função carrega a imagem. Especifica uma combinação das constantes listadas na Tabela 1449.4.

Como a Tabela 1449.1 indica, seus programas podem carregar imagens OEM. Nesses casos, seus programas podem especificar um dos identificadores de imagem listados na Tabela 1449.2

Tabela 1449.2 Identificadores da imagem.

Prefixo	Significado
*OBM_*	Mapas de bits OEM
*OIC_*	Ícones OEM
*OCR_*	Cursores OEM

Como você aprendeu, seus programas podem carregar um dentre vários tipos de imagem dentro da função *LoadImage*. O parâmetro *uTipo* especifica o tipo da imagem, e pode ser um dos valores que a Tabela 1449.3 detalha.

Tabela 1449.3 Os tipos possíveis de imagem.

Valor	Significado
IMAGE_BITMAP	Carrega um mapa de bits
IMAGE_CURSOR	Carrega um cursor
IMAGE_ICON	Carrega um ícone

Como visto, você pode carregar o arquivo de imagem com várias opções para a exibição do arquivo. O parâmetro *fuCarga* precisa ser um ou mais dos valores que a Tabela 1449.4 lista.

Tabela 1449.4 Opções de carga para o arquivo da imagem.

Valor	Significado
LR_DEFAULTCOLOR	O sinalizador padrão significa "não *LR_MONOCHROME*".
LR_CREATEDIBSECTION	Quando o parâmetro *uTipo* especifica *IMAGE_BITMAP*, esse parâmetro faz a função retornar um mapa de bits independentes do dispositivo em vez de um mapa de bits compatível. Esse sinalizador é útil para carregar um mapa de bits sem mapeá-lo para as cores do dispositivo de exibição.

Tabela 1449.4 Opções de carga para o arquivo da imagem. (*Continuação*)

Valor	Significado
LR_DEFAULTSIZE	Usa a largura ou altura que os valores da métrica do sistema especifica para os cursores ou ícones se a chamada define os valores para *cxDesejado* ou *cyDesejado* como 0. Se esse sinalizador não for especificado e a chamada definir os valores para *cxDesejado* e *cyDesejado* como zero, a função usará o tamanho do recurso atual. Se o recurso contiver múltiplas imagens, a função usará o tamanho da primeira imagem.
LR_LOADFROMFILE	Carrega a imagem do arquivo que o parâmetro *lpszNome* especifica. Se esse sinalizador não estiver especificado, *lpszNome* será o nome do recurso.
LR_LOADMAP3DCOLORS	Pesquisa na tabela de cores a imagem e substitui os tons de cinza com a cor tridimensional correspondente, como mostrado na Tabela 1449.5.
LR_LOADTRANSPARENT	Lê o valor de cor do primeiro pixel na imagem e substitui o item correspondente na tabela de cores com a cor da janela padrão (*COLOR_WINDOW*). Todos os pixels na imagem que usam esse item se tornam a cor da janela padrão. Esse valor se aplica somente para imagens que têm tabelas de cores correspondentes. Se *fuCarga* incluir ambos os valores *LR_LOADTRANSPARENT* e *LR_LOADMAP3DCOLORS*, *LRLOADTRANSPARENT* terá precedência. No entanto, a função *fuCarga* substituirá o item da tabela de cores com *COLOR_3DFACE* em vez de *COLOR_WINDOW*.
LR_MONOCHROME	Carrega a imagem em branco e preto.
LR_SHARED	Compartilhará o indicativo da imagem se um ou mais programas carregarem a imagem múltiplas vezes simultaneamente. Se você não definir *LR_SHARED*, uma segunda chamada à *LoadImage* para o mesmo recurso carregará a imagem novamente e retornará um indicativo diferente. Não use LR_SHARED para as imagens que têm tamanho não-padrão que podem mudar após a carga ou que seu programa carrega de um arquivo.

Quando você carregar um arquivo de imagem como um arquivo tridimensional, a API do Windows mapeará as cores para você. O Windows substituirá cada cor na coluna da esquerda da Tabela 1449.5 com a cor na coluna direita da tabela.

Tabela 1449.5 Valores de mapa de 3D para ***LoadImage***.

Cor	Substituída por
Cinza Esc RGB(128,128,128)	*COLOR_3DSHADOW*
Cinza, RGB(192,192,192)	*COLOR_3DFACE*
Cinza Cl, RGB(223,223,223)	*COLOR_3DLIGHT*

Como o texto indica, seus programas podem usar a função *LoadImage* em qualquer situação onde poderiam usar *LoadIcon*, *LoadBitmap* ou *LoadCursor*.

1450 COMPREENDENDO A E/S EM ARQUIVOS NO WINDOWS

Em todas as seções anteriores deste livro, você aprendeu sobre a E/S em arquivo. Você aprendeu como executar E/S de arquivo em C e em C++. Na seção a seguir, você aprenderá os fundamentos sobre a E/S em arquivos no Windows.

O conceito tradicional de um arquivo é um bloco de dados em um dispositivo de armazenamento. Um único identificador conhecido como nome de arquivo identifica o bloco de dados. No ambiente do DOS, os programas geralmente gravam arquivos nas unidades de disco. Por outro lado, para propósitos de E/S, a API

Win32 trata as canalizações nomeadas, os recursos de comunicações, os dispositivos de disco, a entrada e saída no console, ou o arquivo tradicional em disco como um "arquivo". Cada tipo diferente de arquivo é o mesmo no nível de base, mas cada tipo de arquivo tem suas próprias propriedades e limitações As funções de arquivo da API Win32 permitem que seus programas acessem arquivos independentemente do sistema de arquivo ou do tipo de dispositivo subjacente. No entanto, como você pode imaginar, as capacidades de um arquivo variam de um sistema de arquivo para outro e de um tipo de dispositivo para outro.

Apresentando as Canalizações, Recursos, Dispositivos e Arquivos 1451

Como visto na dica anterior, o Windows suporta entrada e saída no estilo de arquivo em diversos dispositivos. Em dicas subseqüentes, você aprenderá não somente sobre entrada e saída de arquivo em um arquivo em disco padrão, mas também aprenderá alguns fundamentos de entrada e saída de arquivo em outros dispositivos É útil conhecer alguns dos dispositivos mais comuns que você encontrará ao trabalhar com os programas Windows. A Tabela 1451 lista alguns dos dispositivos mais comuns.

Tabela 1450 Dispositivos comuns e seus usos.

Dispositivo	Uso Mais Comum
Arquivo	Armazenamento persistente de dados.
Diretório	Atributos e compressão de arquivo.
Unidade Lógica de Disco	Formatação.
Unidade Física de Disco	Acesso à Tabela de Partição.
Porta Serial	Transmissão de dados por uma linha telefônica.
Porta Paralela	Transmissão de dados para uma impressora.
Mailslot	Transmissão de dados de um-para-muitos, normalmente em uma rede para uma máquina baseada no Windows.
Canalização nomeada	Transmissão de dados de um-para-um, normalmente em uma rede para uma máquina baseada no Windows.
Canalização anônima	Transmissão de dados de um-para-um em uma única máquina (nunca em uma rede).
Soquete	Transmissão de dados de datagrama ou stream (fluxo, ou canal), geralmente sobre uma rede para qualquer máquina que suporte soquetes (a máquina precisa estar rodando o Windows).
Console	Um buffer de tela de janela de texto.

Como você descobrirá, o Win32 tenta ocultar as diferenças entre dispositivos do usuário tanto quanto possível. Em outras palavras, se você abrir um mailslot e um arquivo, o Windows geralmente permitirá que você leia ou grave em ambos os dispositivos com funções similares.

No entanto, você deve observar, que além de *CreateFile*, *ReadFile* e *WriteFile*, o Windows fornece uma coleção extensa de funções específicas de dispositivo que permite ao seus programas gerenciar de perto as características específicas dos dispositivos. Por exemplo, não faz sentido definir uma taxa em bauds quando você usa uma canalização para comunicar-se, embora fará sentido fazer isso quando você usar uma porta de comunicações. Portanto, a função *SetCommConfig* trabalhará com um dispositivo de porta serial, mas não trabalhará corretamente com uma canalização nomeada. Por essa razão, a maior parte das dicas a seguir enfocará o uso generalizado das funções *CreateFile*, *ReadFile* e *WriteFile*, em vez de aplicação específica de uma função para um dado dispositivo. Referencie a documentação do seu compilador e do dispositivo para saber maiores informações sobre a comunicação com um certo dispositivo específico.

Usando a Função CreateFile para Abrir Arquivos 1452

Como você aprendeu, a API Win32 suporta muitos tipos de dispositivo e normalmente permitirá que seus programas manipulem arquivos em todos os tipos de dispositivo. Para criar um arquivo em qualquer dispositivo,

seus programas geralmente usarão a função *CreateFile* da API Win32. A função *CreateFile* cria ou abre os seguintes objetos e retornam um indicativo que seus programas podem então usar para acessar o objeto:

- Arquivos
- Canalizações
- Mailslots
- Recursos de comunicação
- Dispositivos de disco (Windows NT somente)
- Consoles
- Diretórios (abrir somente)

Você usará a função *CreateFile* dentro de seus programas como mostrado no seguinte protótipo:

```
HANDLE CreateFile(
   LPCTSTR lpNomeArq,        // ponteiro para o nome do arquivo
   DWORD dwAcessoDesejado,   // modo de acesso (leitura // somente)
   DWORD dwModoCompart,      // modo de compartilhamento
   LPSECURITY_ATTRIBUTES lpAtribSeg, // atributos de segurança
   DWORD dwCriacaoDistr,     // como criar
   DWORD dwSinalizEAtrib,    // atributos do arquivo
   HANDLE hGabaritoArq       // indicativo para arquivo de gabarito
```

A função *CreateFile* fornece aos seus programas controle significativo sobre o arquivo que você criar. A Tabela 1452.1 lista os parâmetros para a função *CreateFile*.

Tabela 1452.1 Parâmetros para a função ***CreateFile***.

Parâmetros	Descrição
lpNomeArq	Aponta para uma string terminada por *NULL* que especifica o nome do objeto (arquivo, canalização, mailslot, recurso de comunicação, dispositivo de disco, console ou diretório) que a função *CreateFile* deve criar ou abrir. Se **lpNomeArq* for um caminho, haverá um limite no tamanho da string padrão de *MAX_PATH* caracteres. Esse limite se relaciona como a função *CreateFile* analisa os caminhos.
dwAcessoDesejado	Especifica o tipo de acesso para o objeto Um aplicativo pode obter acesso de leitura, acesso de escrita, acesso de leitura e escrita, ou acesso de consulta de dispositivo. Esse parâmetro pode ser qualquer combinação dos valores que a Tabela 1452.2 detalha.
dwModoCompart	Conjunto de bits sinalizadores que especifica como os programas poderão compartilhar o objeto. Se *dwModoCompart* for zero, os programas não poderão compartilhar o objeto. As operações de abertura subseqüente do objeto falharão, até que o programa que está usando o objeto feche o indicativo. Para compartilhar o objeto, use uma combinação de um ou mais dos valores que a Tabela 1452.3 detalha.
lpAtribSeg	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribSeg* for *NULL*, os processos-filho não poderão herdar o indicativo. Windows NT. o membro *lpSecurityDescriptor* da estrutura especifica um descritor de segurança para o objeto. Se *lpAtribSeg* for *NULL*, o objeto recebe um descritor de segurança padrão. O sistema de arquivo alvo precisa suportar segurança em arquivos e diretórios para esse parâmetro ter um efeito nos arquivos. Windows 95: *CreateFile* ignora o membro *lpSecurityDescriptor*.

Tabela 1452.1 Parâmetros para a função ***CreateFile***. (*Continuação*)

Parâmetros	Descrição
dwCriacaoDist	Especifica qual ação tomar em arquivos que existem e qual ação tomar quando arquivos não existirem. Esse parâmetro precisa ser um dos valores que a Tabela 1452.4 lista.
dwSinalizEAtrib	Especifica os atributos de arquivo e os sinalizadores para o arquivo. Qualquer combinação dos atributos que a Tabela 1452.5 lista é aceitável para o parâmetro *dwSinalizEAtrib*, exceto que todos os outros atributos de arquivo anulam *FILE_ATTRIBUTE_NORMAL*. Se a função *CreateFile* abrir o lado cliente de uma canalização nomeada, o parâmetro *dwSinalizEAtrib* também poderá conter as informações Security Quality of Service (SQOS, ou Qualidade de Segurança do Serviço). A dica a seguir explicará a abertura de canalizações nomeadas em detalhes. Quando o aplicativo de chamada especifica o sinalizador *SECURITY_SQOS_PRESENT*, o parâmetro *dwSinalizEAtrib* pode conter um ou mais dos valores que a Tabela 1453.1, mais à frente, listará.
hGabaritoArq	Especifica um indicativo com acesso *GENERIC_READ* para um arquivo de gabarito. O arquivo de gabarito fornece atributos de arquivo e atributos estendidos para o arquivo que a função *CreateFile* está criando. Sob o Windows 95, esse valor precisa ser *NULL*. Se você fornecer um indicativo sob o Windows 95, a chamada falhará.

Como você aprendeu na Tabela 1452.1, seus programas podem especificar o nível de acesso desejado para os arquivos que abrem com *CreateFile* dentro do parâmetro *dwAcesso*. Os valores possíveis para o parâmetro *dwAcesso* são mostrados na Tabela 1452.2.

Tabela 1452.2 Valores possíveis para o membro ***dwAcesso***.

Valor	Significado
0	Especifica acesso de consulta de dispositivo para o objeto. Um aplicativo pode consultar os atributos do dispositivo sem acessar o dispositivo.
GENERIC_READ	Especifica o acesso de leitura para o objeto. Os programas podem ler dados do arquivo e as chamadas da API podem mover o ponteiro do arquivo. Combina com *GENERIC_WRITE* para o acesso de leitura e escrita.
GENERIC_WRITE	Especifica o acesso de escrita para o objeto. Os programas podem escrever dados do arquivo, e as chamadas da API podem mover o ponteiro do arquivo. Combina com *GENERIC_READ* para o acesso de leitura e escrita.

Além dos níveis de acesso, seus programas podem especificar um modo de compartilhamento para o arquivo dentro do parâmetro *dwModoCompart*. A Tabela 1452.3 lista os possíveis valores para o parâmetro *dwModoCompart*.

Tabela 1452.3 Valores possíveis para o parâmetro ***dwModoCompart***.

Valor	Significado
FILE_SHARE_DELETE	Windows NT somente: As operações de abertura subseqüentes no objeto serão bem-sucedidas somente se o processo de abertura requisitar o acesso de exclusão.
FILE_SHARE_READ	As operações de abertura subseqüentes no objeto serão bem-sucedidas somente se o processo de abertura solicitar o acesso de leitura.
FILE_SHARE_WRITE	As operações de abertura subseqüentes no objeto serão bem sucedidas somente se o processo de abertura solicitar o acesso de gravação.

O parâmetro *dwCria* determina a açao que o Windows deverá tomar quando o arquivo existir ou não existir. O parâmetro precisa ser um dos valores na Tabela 1452.4.

Tabela 1452.4 Valores possíveis para o parâmetro ***dwCria***.

Valor	Significado
CREATE_NEW	Cria um novo arquivo. A função falha se o arquivo especificado já existe.
CREATE_ALWAYS	Cria um novo arquivo. A função sobrescreve o arquivo se ele já existe.
OPEN_EXISTING	Abre o arquivo A função falha se o arquivo não existe
OPEN_ALWAYS	Abre o arquivo, se ele existe. Se o arquivo não existe, a função cria o arquivo como se a *dwCriacaoDistr* fosse *CREATE_NEW*.
TRUNCATE_EXISTING	Abre o arquivo. Uma vez aberto, o Windows trunca o arquivo para que seu tamanho fique zerado. O processo de chamada precisa abrir o arquivo com pelo menos acesso *GENERIC_WRITE*. A função falha se o arquivo ainda não existir.

Quando você criar qualquer arquivo dentro do Windows, o sistema operacional Windows associará atributos com esse arquivo. Você poderá especificar sinalizadores e atributos para cada novo arquivo que criar. A Tabela 1452.5 lista os possíveis sinalizadores e atributos.

Tabela 1452.5 O sinalizador possível e os valores de atributos para o parâmetro ***dwSinalizEAtrib***.

Atributo	Significado
FILE_ATTRIBUTE_ARCHIVE	Marca o arquivo para backup. Os aplicativos usam esse atributo para marcar os arquivos para backup ou remoção.
FILE_ATTRIBUTE_COMPRESSED	O arquivo ou diretório está compactado. Para um arquivo, isso significa que todos os dados no arquivo estão compactados. Para um diretório, significa que a compactação é o normal para arquivos e subdiretórios recém-criados.
FILE_ATTRIBUTE_HIDDEN	O arquivo está oculto (não aparece na listagem normal do diretório).
FILE_ATTRIBUTE_NORMAL	O arquivo não tem outros atributos definidos. Esse atributo é válido somente se seu programa o usa sozinho quando o programa chama *CreateFile*.
FILE_ATTRIBUTE_OFFLINE	Os dados do arquivo não estão disponíveis imediatamente. Indica que os dados do arquivo foram fisicamente movidos para armazenagem off-line.
FILE_ATTRIBUTE_READONLY	O arquivo é de leitura somente. Os aplicativos podem ler o arquivo mas não podem gravar nele ou excluí-lo.
FILE ATTRIBUTE SYSTEM	O arquivo faz parte do sistema operacional, ou só o sistema pode usá-lo.
FILE_ATTRIBUTE_TEMPORARY	Um processo está usando o arquivo para armazenagem temporária. Um aplicativo deverá excluir um arquivo temporário assim que não precisar mais dele.
FILE_FLAG_WRITE_THROUGH	Instrui o sistema a gravar usando qualquer cache intermediário e a ir diretamente ao disco. O Windows ainda poderá usar o cache para as operações de gravação, mas não poderá descarregá-las quando quiser.

Tabela 1452.5 O sinalizador possível e os valores de atributos para o parâmetro ***dwSinalizEAtrib.*** (*Continuação*)

Atributo	Significado
FILE_FLAG_OVERLAPPED	Instrui o sistema a inicializar o objeto, para que as operações que requerem uma quantidade de tempo de processamento significativa retornem *ERROR_IO_PENDING* Quando o sistema operacional finalizar a operação, o Windows deixará o evento especificado dentro da estrutura *OVERLAPPED* com o estado sinalizado. Quando você especifica *FILE FLAG_OVERLAPPED*, as funções *ReadFile* e *WriteFile* precisam especificar uma estrutura *OVERLAPPED*. Isto é, quando você especificar *FILE_FLAG_OVERLAPPED*, um aplicativo precisa executar leitura e gravação sobrecarregadas. Quando você especifica *FILE_FLAG_OVERLAPPED*, o sistema não mantém o ponteiro de arquivo. O processo chamador precisa passar a posição do arquivo como parte do parâmetro *lpOverlapped* (apontando para uma estrutura *OVERLAPPED*) para as funções *ReadFile* e *WriteFile*. Esse sinalizador também permite que um processo execute mais de uma operação simultaneamente com um único indicativo (uma operação de leitura e gravação simultânea, por exemplo).
FILE_FLAG_NO_BUFFERING	Instrui o sistema operacional a abrir o arquivo sem bufferização ou cache intermediário. Quando combinado com *FILE_FLAG_OVERLAPPED*, o sinalizador dá máximo desempenho assíncrono porque a E/S não depende das operações síncronas do gerenciador de memória. No entanto, algumas operações de E/S serão mais demoradas porque o sistema operacional não está armazenando dados no cache. Um aplicativo precisa atender a certos requisitos ao trabalhar com arquivos que você abrir com *FILE FLAG_NO_BUFFERING*. O acesso ao arquivo precisa iniciar em deslocamentos de byte dentro do arquivo que sejam múltiplos inteiros do tamanho do setor do volume. Por exemplo, se o tamanho do setor for 512 bytes, um aplicativo poderá solicitar leituras e gravações de 512, 1024 ou 2048 bytes, mas não de 335, 981 ou 7171 bytes. Os endereços de buffer para as operações de leituras e gravações precisam estar alinhados em endereços de memória que são múltiplos inteiros do tamanho do setor do volume. Um modo de alinhar buffers em inteiros múltiplos do tamanho do setor do volume é usar *VirtualAlloc* para alocar os buffers. Ele aloca memória que está alinhada em endereços que são inteiros múltiplos do tamanho da página de memória do sistema operacional. Como tanto as páginas de memória quanto os tamanhos do setor do volume são potências de 2, essa memória também está alinhada em endereços que são inteiros múltiplos do tamanho de setor de um volume. Um aplicativo pode chamar a função *GetDiskFreeSpace* para determinar o tamanho do setor de um volume.
FILE_FLAG_RANDOM_ACCESS	Indica que o processo acessará o arquivo aleatoriamente. O sistema pode usar esse atributo como uma dica para otimizar o cache de arquivo para aquele arquivo.

Tabela 1452.5 O sinalizador possível e os valores de atributos para o parâmetro ***dwSinalizEAtrib***. (*Continuação*)

Atributo	Significado
FILE_FLAG_SEQUENTIAL_SCAN	Indica que o processo acessará o arquivo seqüencialmente do início ao fim. O sistema pode usar esse atributo como uma dica para otimizar o cache de arquivo para esse arquivo. Se o aplicativo mover o ponteiro de arquivo para acesso aleatório, o cache mínimo poderá não ocorrer; no entanto, a operação correta ainda está garantida. Especificar esse sinalizador pode aumentar o desempenho para os aplicativos que usam o acesso seqüencial para ler arquivos grandes. Os ganhos de desempenho podem ser até mais notáveis para os aplicativos que lêem grandes arquivos seqüencialmente na maior parte das vezes, mas que ocasionalmente saltam pequenos intervalos de bytes.
FILE_FLAG_DELETE_ON_CLOSE	Indica que o sistema operacional deverá excluir o arquivo imediatamente após todos os seus indicativos terem sido fechados, não apenas o indicativo para o qual você especificou *FILE_FLAG_DELETE_ON_CLOSE*. As solicitações de abertura subseqüentes para o arquivo falharão, a não ser que o processo de chamada use o sinalizador *FILE_SHARE_DELETE*.
FILE_FLAG_BACKUP_SEMANTICS	Windows NT somente: Indica que o arquivo está sendo aberto ou criado para uma operação de backup ou de restauração. O sistema operacional garante que o processo chamador anula as verificações de segurança do arquivo, desde que ele tenha a permissão necessária para fazer isso. As permissões relevantes são *SE_BACKUP_NAME* e *SE_RESTORE_NAME*. Você também pode definir esse sinalizador para obter um indicativo para um diretório. Um processo pode passar um indicativo de diretório para algumas funções Win32 em lugar de um indicativo de arquivo.
FILE_FLAG_POSIX_SEMANTICS	Indica que o processo deverá acessar o arquivo de acordo com as regras POSIX. Isso inclui permitir múltiplos arquivos com nomes, diferindo somente na caixa das letras, para os sistemas de arquivo que suportam essa nomeação. Tenha cuidado quando você usar essa opção porque os aplicativos escritos para o MS-DOS ou Windows poderão não ser capazes de acessar os arquivos que você criar com esse sinalizador.

Se a função for bem-sucedida, ela retornará um indicativo para o arquivo especificado. Se o arquivo especificado existir antes da chamada à função, e *dwCriacaoDist* for *CREATE_ALWAYS* ou *OPEN_ALWAYS*, uma chamada à *GetLastError* retornará *ERROR_ALREADY_EXISTS* (mesmo que a função tenha sido bem-sucedida). Se o arquivo não existir antes da chamada, *GetLastError* retornará zero. Se a função falhar, ela retornará *INVALID_HANDLE_VALUE*.

Como notado anteriormente, especificar zero para *dwAcessoDesejado* permite que um aplicativo consulte os atributos do dispositivo sem, na verdade, acessar o dispositivo. Esse tipo de consulta é útil, por exemplo, se um aplicativo quiser determinar o tamanho de uma unidade de disquete e os formatos que ele suporta sem ter um disquete na unidade.

O CD-ROM que acompanha este livro inclui o programa *First_File.cpp*, que cria o arquivo *file.dat* e grava uma string nele. Quando o usuário selecionar *Testar!*, o programa lerá a string do arquivo e exibirá a string dentro de uma caixa de mensagem.

USANDO CREATEFILE COM DIFERENTES DISPOSITIVOS 1453

Como você aprendeu, seus programas podem usar a função *CreateFile* para criar arquivos em uma grande variedade de dispositivos. No entanto, existem diferenças significativas em como *CreateFile* opera dependendo do dispositivo que seu programa passa para a função. Os parâmetros a seguir descrevem algumas das variações no processamento da função *CreateFile* que depende do dispositivo.

Quando você usar a função *CreateFile* para criar um novo arquivo, a função *CreateFile* executa as seguintes ações:

- Combina os atributos de arquivo e os sinalizadores que *dwSinalizEAtrib* especifica com *FILE_ATTRIBUTE_ARCHIVE*.
- Define o tamanho do arquivo como 0.
- Copia os atributos estendidos que o gabarito do arquivo fornece para o novo arquivo se você especifica o parâmetro *hGabaritoArq*.

Quando a função *CreateFile* abre um arquivo existente, ela executa as seguintes ações:

- Combina os sinalizadores de arquivo que *dwSinalizEAtrib* especifica com os atributos de arquivos existentes. A função *CreateFile* ignora os atributos de arquivo que *dwSinalizEAtrib* especifica.
- Define o tamanho do arquivo de acordo com o valor de *dwCriacaoDist*.
- Ignora o parâmetro *hGabaritoArq*.
- Ignora o membro *lpSecurityDescriptor* da estrutura *SECURITY_ATTRIBUTES* se o parâmetro *lpAtribSeg* não é *NULL*. *CreateFile* não usa os outros membros da estrutura. O membro *bInheritHandle* é o único modo de indicar se outro processo pode herdar o indicativo de arquivo.

Se você tentar criar um arquivo em uma unidade de disquete que não tenha um disquete ou abrir um arquivo em uma unidade de CD-ROM que não tenha um CD, o sistema exibirá uma caixa de mensagem pedindo ao usuário para inserir um disquete ou um CD, respectivamente. Para evitar que o sistema exiba essa caixa de mensagem, chame a função *SetErrorMode* com *SEM_FAILCRITICALERRORS*. Para obter maiores informações sobre *SetErrorMode*, verifique a documentação on-line do seu compilador.

Se *CreateFile* abrir a ponta cliente de uma canalização nomeada, a funçao usará qualquer ocorrência da canalização nomeada que esteja no estado de audição. O processo de abertura pode duplicar o indicativo quantas vezes forem necessárias, mas, após *CreateFile* tiver aberto a ponta cliente, outro cliente não poderá abrir a ocorrência da canalização nomeada. O acesso que você especificar quando *CreateFile* abrir uma canalização precisará ser compatível com o acesso que você especificar no parâmetro *dwModoAbertura* da função *CreateNamedPipe*. Quando você especificar segurança em conexão com a abertura de uma canalização nomeada, seu programa também deverá especificar um ou mais dos sinalizadores de segurança que a Tabela 1453.1 detalha.

Tabela 1453.1 Sinalizadores de segurança para uso ao criar uma canalização nomeada.

Valor	Significado
SECURITY_ANONYMOUS	Especifica que o arquivo imitará o cliente no nível Anonymous.
SECURITY_IDENTIFICATION	Especifica que o arquivo imitará o cliente no nível Identification.
SECURITY IMPERSONATION	Especifica que o arquivo imitará o cliente no nível Impersonation.
SECURITY_DELEGATION	Especifica que o arquivo imitará o cliente no nível Delegation.
SECURITY_CONTEXT_TRACKING	Especifica que o modo de controle de segurança é dinâmico. Se você não especificar esse sinalizador, Security Tracking Mode será estático
SECURITY_EFFECTIVE_ONLY	Especifica que somente os aspectos habilitados do contexto de segurança cliente estarão disponíveis para o servidor. Se você não especificar esse sinalizador, todos os aspectos do contexto de segurança do cliente estarão disponíveis. Esse sinalizador permite que o cliente limite os grupos e privilégios que um usuário pode usar ao imitar o cliente.

Se a função *CreateFile* abrir a ponta cliente de um mailslot, a função retornará *INVALID_HANDLE_VALUE* se o cliente de mailslot tentar abrir um mailslot local antes que o servidor de mailslot o tenha criado usando a função *CreateMailSlot*.

A função *CreateFile* pode criar um indicativo para um recurso de comunicações, tal como a porta serial COM1. Para recursos de comunicação, o parâmetro *dwCriacaoDist* precisa ser *OPEN_EXISTING*, e o parâmetro *hGabarito* precisa ser *NULL*. Você pode especificar acesso de leitura e gravação ou leitura-gravação e pode abrir o indicativo para E/S sobreposta.

Sob o Windows NT, você pode usar a função *CreateFile* para abrir uma unidade de disco ou uma partição em uma unidade de disco. A função retorna um indicativo para o dispositivo de disco. Seus programas podem, mais tarde, usar esse indicativo com a função *DeviceIOControl*. A chamada precisa atender aos seguintes requisitos para que a chamada seja bem-sucedida:

- O chamador precisa ter privilégios administrativos para a operação ser bem-sucedida em uma unidade de disco rígido.
- A string *lpNomeArq* precisa estar na forma \\.*unidfisicax* para abrir o disco rígido *x*. Os números de discos rígidos iniciam em 0. Por exemplo, \\.*unidfisica2* obtém um indicativo para a terceira unidade *física no computador do usuário*.
- A string *lpNomeArq* deve ser \\.\x: para abrir uma unidade de disquete *x* ou uma partição em um disco rígido. Por exemplo, \\.*A* obtém um indicativo para a unidade A no computador do usuário e \\.\C obtém um indicativo para a unidade C no computador do usuário.
- Sob o Windows 95, essa técnica não funciona para abrir uma unidade lógica. No Windows 95, especificar uma string nesta forma faz *CreateFile* retornar um erro.
- O parâmetro *dwCriacaoDist* precisa ter o valor *OPEN_EXISTING*.
- Ao abrir uma unidade de disquete ou uma partição em um disco rígido, você precisa definir o sinalizador *FILE_SHARE_WRITE* no parâmetro *dwModoCompart*.

A função *CreateFile* pode criar um indicativo para entrada no console (*CONIN$*). Se o processo tiver um indicativo aberto para entrada no console como um resultado da herança ou duplicação, o processo também pode criar um indicativo para o buffer de tela ativo (*CONOUT$*). Para os indicativos de console, defina os parâmetro *CreateFile* conforme a Tabela 1453.2 detalha.

Tabela 1453.2 Valores de parâmetro para ***CreateFile*** quando você criar um console.

Parâmetros	Valor
lpNomeArq	Use o valor *CONIN$* para especificar entrada no console e o valor *CONOUT$* para especificar a saída no console. *CONIN$* obtém um indicativo para o buffer de tela ativo, mesmo se *SetStd_Handle* redirecionou o indicativo de saída padrão. Para obter o indicativo de entrada padrão, use *GetStdHandle*. *CONOUT$* obtém um indicativo para o buffer de tela ativo, mesmo se *SetStdHande* redirecionou o indicativo de saída padrão. Para obter o indicativo de saída padrão, use *GetSetHandle*.
dwAcessoDesejado	A Microsoft recomenda que você use somente *GENERIC_READ* | *GENERIC_WRITE*, mas seus programas podem usar uma das duas para limitar o acesso.
dwModoCompart	Se o processo chamador herdou o console ou se um processo-filho pode ser capaz de acessar o console, esse parâmetro precisa ser *FILE_SHARE_READ* | *FILE_SHARE_WRITE*.
lpAtribSegur	Se você quiser que o processo-filho herde o console, o membro *hInheritHandle* da estrutura *SECURITY_ATTRIBUTES* precisará ser *True*.
dwCriacaoDist	Você deverá especificar *OPEN_EXISTING* quando usar *CreateFile* para abrir o console.
dwSinalizEAtrib	Ignorado.
bGabaritoArq	Ignorado.

A Tabela 1453.3 mostra os efeitos de vários valores de *dwAcessoDesejado* e *lpNomeArq* quando *lpNomeArq* é deixado como *CON*.

Tabela 1453.3 Os efeitos de acessar os valores ao abrir um console.

Valores	Resultado
GENERIC_READ	Abre o console para entrada.
GENERIC_WRITE	Abre o console para saída.
GENERIC_READ | *GENERIC_WRITE*	Faz *CreateFile* falhar.

Um aplicativo não pode criar um diretório com *CreateFile*. Em vez disso, para criar um diretório, o aplicativo precisa chamar *CreateDirectory* ou *CreateDirectoryEx*. No entanto, sob o Windows NT, você pode definir o sinalizador *FILE_FLAG_BACKUP_SEMANTICS* para obter um indicativo para um diretório. Seus programas podem passar um indicativo de diretório para algumas funções do Win32 em lugar de um indicativo de arquivo. Alguns sistemas de arquivo, tais como NTFS, suportam compressão para arquivos e diretórios individuais. Em volumes formatados para tal sistema e arquivo, um novo diretório herda o atributo de compressão do diretório-pai.

USANDO INDICATIVOS DE ARQUIVO 1454

Exatamente como o DOS e o UNIX, o Windows atribui um indicativo de arquivo para cada arquivo que seus programas abrirem ou criarem. Um indicativo de arquivo é um identificador único que um aplicativo usa dentro das funções que acessam arquivos. Os indicativos são válidos até que seus aplicativos fechem os arquivos com a função *CloseHandle*, que fecha o arquivo e descarrega os buffers no disco. Quando um aplicativo inicia, ele herda todos os indicativos do arquivo aberto a partir do processo que iniciou o aplicativo, desde que o processo-pai tenha aberto o arquivo e permitido a herança. Se o processo-pai abriu os arquivos sem permitir a herança, o aplicativo não herdará os indicativos de arquivo abertos.

Como você aprendeu, um aplicativo pode abrir indicativos de arquivo para entrada e saída no console. Em vez de um nome de arquivo, o aplicativo passa a string *CONIN$* para a função *CreateFile* como o nome de arquivo para a entrada no console, e *CONOUT$* como o nome de arquivo para a saída no console.

REVISITANDO OS PONTEIROS DE ARQUIVO 1455

Como você já aprendeu na seção Arquivo, Diretórios e Discos, quando seus aplicativos abrirem um arquivo, os sistemas operacionais geralmente colocarão um ponteiro de arquivo no início do arquivo. O Windows não é exceção. O ponteiro de arquivo marca a posição atual no arquivo onde a próxima operação de leitura ou gravação terá lugar. À medida que seus programas lerem ou gravarem cada byte em um arquivo, o Windows avançará o ponteiro de arquivo para o próximo byte. Um aplicativo também pode mover a posição do ponteiro de arquivo com a função *SetFilePointer*. Você usará a função *SetFilePointer* dentro de seus programas, como mostrado no protótipo a seguir:

```
DWORD SetFilePointer(
   HANDLE  hArquivo,          // indicativo do arquivo
   LONG  lDistanciaAMover,    // número de bytes a mover o ponteiro
   PLONG lpDistanciaAMoverAlto,  // endereço da palavra de
                        // alta ordem da distância a mover
  DWORD dwMetodoMover             // como mover
);
```

A função *SetFilePointer* usa os parâmetros que a Tabela 1455.1 detalha.

Tabela 1455.1 Parâmetros para a função ***SetFilePointer***

Parâmetro	Descrição
hArquivo	Identifica o arquivo cujo ponteiro de arquivo é a função a mover. O indicativo de arquivo precisa ter o acesso *GENERIC_READ* ou *GENERIC_WRITE* para o arquivo.
lDistanciaAMover	Especifica o número de bytes a mover o ponteiro de arquivo. Um valor positivo move o ponteiro para a frente no arquivo, e um valor negativo move-o para trás.
lpDistanciaAMoverAlto	Aponta para a palavra mais significativa da distância de 64 bits a mover. Se o valor desse parâmetro for *NULL*, *SetFilePointer* poderá operar somente com arquivos cujo tamanho máximo é $(2^{32}-2)$. Se você especificar esse parâmetro, o tamanho máximo do arquivo será $(2^{64}-2)$. Esse parâmetro também recebe a palavra de alta ordem do novo valor do ponteiro de arquivo.
dwMetodoMover	Especifica o ponto inicial para a movimentação do ponteiro de arquivo. Esse parâmetro pode ser um dos valores que a Tabela 1455.2 detalha.

Existem vários métodos diferentes que você pode instruir o Windows a usar para mover o ponteiro de arquivo. A Tabela 1455.2 lista os métodos predefinidos de movimentação de ponteiro do Windows.

Tabela 1455.2 Os métodos de movimentação possíveis para ***SetFilePointer***.

Valor	Significado
FILE_BEGIN	O ponto inicial é zero ou o início do arquivo. Se *FILE_BEGIN* estiver especificado, a função interpretará *lDistanciaAMover* como uma localização não-sinalizada para o novo ponteiro de arquivo.
FILE_CURRENT	O valor atual do ponteiro de arquivo é o ponto inicial.
FILE_END	A posição atual de final do arquivo é o ponto inicial.

Se a função *SetFilePointer* for bem-sucedida, ela retornará a palavra dupla (uma *double-word*) menos significativa do novo ponteiro de arquivo, e, se *lpDistanciaAMoverAlto* não for *NULL*, a função colocará a palavra dupla mais significativa do novo ponteiro de arquivo no valor *long* apontado por esse parâmetro. Se a função falhar e *lpDistanciaAMoverAlto* for *NULL*, ela retornará *0xFFFFFFFF*. Se a função falhar e *lpDistanciaAMoverAlto* não for *NULL*, a função retornará *0xFFFFFFFF*, e *GetLastError* retornará um valor diferente de *NO_ERROR*.

Você não pode usar a função *SetFilePointer* com um indicativo para um dispositivo que não faz busca (a operação *Seek*), tal como um dispositivo de canalização ou de comunicação. Para determinar o tipo de arquivo para *hArquivo*, use a função *GetFileType*. Seja cuidadoso ao definir o ponteiro de arquivo em um aplicativo multiencadeado. Um aplicativo cujos encadeamentos compartilham um indicativo de arquivo, atualizam o ponteiro e lêem o arquivo que precisa usar um objeto de seção crítica ou um objeto mutex para proteger as atualizações de ponteiro de arquivo. Se o indicativo de arquivo *hArquivo* foi aberto com o sinalizador *FILE_FLAG_NO_BUFFERING* ligado, um aplicativo pode mover o ponteiro de arquivo somente para *posições alinhadas em setor*. Uma posição alinhada em setor é uma posição que é um número inteiro múltiplo do tamanho do setor da unidade de disco. Um aplicativo pode chamar a função *GetDiskFreeSpace* para obter um tamanho de setor do disco. Se um aplicativo chamar *SetFilePointer* com valores de distância a mover que resultam em uma posição que não está alinhada em um setor e um indicativo que o programa abriu originalmente aberto com *FILE_FLAG_NO_BUFFERING*, a função falhará e *GetLastError* retornará *ERROR_INVALID_PARAMETER*.

Nota: SetFilePointer é similar à função lseek, já explicada anteriormente na Dica 408, e à função fseek, já explicada na Dica 450.

1456 USANDO WRITEFILE PARA ESCREVER EM UM ARQUIVO

Seus programas podem abrir arquivo com a função *CreateFile*. Se você criou um arquivo com acesso de escrita, seus programas podem então usar a função *WriteFile* para escrever dados nesse arquivo. A função *WriteFile* escreve dados em um arquivo, e foi projetada tanto para a operação síncrona quanto assíncrona. A função inicia

escrevendo dados no arquivo na posição que o ponteiro de arquivo indica. Após *WriteFile* completar a operação de escrita, ela ajusta o ponteiro de arquivo utilizando o número de bytes que ela realmente escreveu, exceto quando o arquivo estiver aberto com *FILE_FLAG_OVERLAPPED*. Se você criou o indicativo de arquivo para entrada e saída (E/S) sobrepostas, o aplicativo precisa ajustar a posição do ponteiro de arquivo após o final da operação de escrita. Você usará a função *WriteFile* dentro de seus programas, como segue:

```
BOOL WriteFile(
   HANDLE hArq,                  // indicativo para o arquivo onde gravar
   LPCVOID lpBuffer,             // ponteiro para os dados
                                 // a escrever no arquivo
   DWORD nNumBytesAGravar,       // número de bytes a gravar
   2PDWORD lpNumBytesGravados,   // ponteiro para o número de bytes gravados
   LPOVERLAPPED lpSobreposta     // ponteiro para estrutura
                                 // necessária para E/S sobreposta
```

A função *WriteFile* aceita os parâmetros detalhados na Tabela 1456.

Tabela 1456 Parâmetros para a função *WriteFile*

Parâmetro	Descrição
hArquivo	Identifica o arquivo em que a função vai escrever. O indicativo de arquivo precisa ter o acesso *GENERIC_WRITE* ao arquivo. Sob o W.ndows NT, para operações de escrita assíncronas, *hArq* pode ser qualquer indicativo que a função *CreateFile* abre com o sinalizador *FILE_FLAG_OVERLAPPED*, ou um indicativo de soquete que as funções *socket* ou *accept* retornam
lpBuffer	Aponta para o buffer que contém os dados que a função escreverá no arquivo.
nNumBytesGravar	Especifica o número de bytes a gravar no arquivo. Ao contrário do MS-DOS, o Windows NT interpreta um valor zero como especificando uma operação de escrita *NULL*. Uma operação de escrita *NULL* não grava quaisquer bytes, mas modifica a informação de hora do arquivo.
lpNumBytesGravados	Aponta para o número de bytes que *WriteCall* grava. *WriteFile* define esse valor como zero antes de fazer qualquer trabalho ou verificação de erro. Se *lpSobreposta* for *NULL*, *lpNumBytesGravados* não poderá ser *NULL*. Se *lpSobreposta* não for *NULL*, *lpNumBytesGravados* poderá ser *NULL*. Se essa for uma operação de escrita sobreposta, você poderá chamar *GetOverlappedResult* para obter o número de bytes gravados. Se *hArquivo* estiver associado com uma porta de E/S, você poderá chamar *GetQueuedCompletionStatus* para obter o número de bytes gravados.
lpSobreposta	Aponta para uma estrutura *OVERLAPPED*. A chamada da função requer essa estrutura se o processo abriu *hArquivo* com o sinalizador *FILE_FLAG_OVERLAPPED* ligado. Se o processo abriu *hArquivo* com *FILE_FLAG_OVERLAPPED*, o parâmetro *lpSobreposta* não poderá ser *NULL*. Ele precisará apontar para uma estrutura *OVERLAPPED* válida. Se *hArquivo* foi aberto com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* for *NULL*, a função poderá informar incorretamente que a operação de gravação está completa. Se o processo abriu *hArquivo* com o sinalizador *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, a operação de escrita iniciará no deslocamento especificado na estrutura *OVERLAPPED* e *WriteFile* poderá retornar antes de o sistema operacional completar a operação de escrita. Nesse caso, *WriteFile* retorna *FALSE*, e a função *GetLastError* retorna *ERROR_IO_PENDING*. Usar E/S em arquivo sobreposto permite que o processo chamador continue processando enquanto o sistema operacional completa a operação de escrita. O sistema operacional define o evento que você especificar na estrutura *OVERLAPPED* para o estado sinalizado ao final da operação de escrita

Tabela 1456 Parâmetros para a função *WriteFile*. (*Continuação*)

Parâmetro	Descrição
	Se o processo não abrir *hArquivo* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* for *NULL*, a operação de escrita iniciará na posição de arquivo atual, e *WriteFile* não retornará até que o sistema operacional tenha completado a operação.
	Se o processo não abrir *hArquivo* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, a operação de escrita iniciará no deslocamento que você especificou na estrutura OVERLAPPED, e WriteFile não retornará até que o sistema operacional complete a operação de escrita.

Se a função *WriteFile* for bem-sucedida, ela retornará um valor diferente de zero.

Se ela falhar, a função retornará zero. Se outro processo bloquear parte do arquivo e a operação de escrita sobrepuser a porção bloqueada, essa função falhará. Os aplicativos não devem ler ou gravar no buffer de saída que uma operação de escrita esteja usando até que a operação de escrita termine. O acesso prematuro do buffer de saída pode levar à corrupção dos dados gravados a partir desse buffer.

Seus programas podem usar *WriteFile* com um indicativo para saída no console para escrever caracteres no buffer de tela. O modo de console determina o comportamento exato da função Os dados são gravados na posição atual do cursor. O modo de console determina o comportamento exato da função. Os dados são gravados na posição atual do cursor. O sistema operacional atualiza a posição de cursor na tela após a operação de escrita. Ao contrário do sistema operacional MS-DOS, o Windows NT interpreta zero bytes a gravar como especificar uma operação de escrita *NULL*, e *WriteFile* não trunca ou estende o arquivo. Para truncar ou estender um arquivo, use a função *SetEndOfFile*.

Quando um aplicativo usa a função *WriteFile* para escrever em uma canalização, a operação de escrita pode não terminar se o buffer de canalização estiver cheio. Em vez disso, a operação de escrita terminará quando uma operação de leitura (usando a função *ReadFile*) tornar mais espaço de buffer disponível. Se um processo fechou o indicativo de canalização de leitura anônima e *WriteFile* tentar usar o indicativo de canalização de escrita anônima para gravar, a função retornará *False*, e *GetLastError* retornará *ERROR_BROKEN_PIPE*.

A função *WriteFile* pode falhar e retornar *ERROR_INVALID_USER_BUFFER* ou *ERROR_NOT_ENOUGH_MEMORY* sempre que existirem muitas solicitações de E/S assíncronas não-atendidas. Para cancelar todas as operações de E/S assíncronas pendentes, use a função *CancelIO*. Essa função somente cancela as operações que o encadeamento chamador para o indicativo de arquivo especificado emitir. As operações de E/S que o sistema operacional cancela são o resultado de uma chamada à *CancelIO* com o erro *ERROR_OPERATION_ABORTED*.

Se você estiver tentando gravar em uma unidade de disquete que não tem um disquete, o sistema exibirá uma caixa de mensagem pedindo que o usuário repita a operação. Para impedir o sistema de exibir essa caixa de mensagem, chame a função *SetErrorMode* com *SEM_NOOPENFILEERORBOX*. Se *hArquivo* for um indicativo a uma canalização nomeada, os membros *Offset* e *OffsetHigh* da estrutura *OVERLAPPED* apontada por *lpSobreposta* precisam ser zero, ou a função falhará.

O CD-ROM que acompanha este livro inclui o programa *Write_File.cpp*, que abre o arquivo *file.dat* e grava uma string no arquivo quando o exemplo inicia. Quando o usuário selecionar a opção *Testar!*, o programa escreverá outra linha de texto no arquivo, e, depois, exibirá ambas as linhas de texto dentro de uma caixa de mensagem.

1457 USANDO READFILE PARA LER A PARTIR DE UM ARQUIVO

Como visto na dica anterior, seus programas podem usar a função *WriteFile* para escrever em um arquivo a partir de dentro do Windows. Similarmente, seus programas podem usar a função *ReadFile* para ler a partir de um arquivo. A função *ReadFile* lê dados de um arquivo, iniciando na posição que o ponteiro de arquivo indica. Após a função *ReadFile* completar a operação de leitura, o sistema operacional ajusta o ponteiro de arquivo pelo número de bytes realmente lidos, a não ser que o processo crie o indicativo de arquivo com o atributo sobreposto. Se o processo criar o indicativo de arquivo para entrada e saída sobreposta, o aplicativo precisará ajustar a posição do ponteiro de arquivo após a operação de leitura. Você usará a função *ReadFile* como mostrado no protótipo a seguir:

```
BOOL ReadFile(
   HANDLE hArquivo,      // indicativo do arquivo a ler
   LPVOID lpBuffer,      // endereço do buffer que recebe os dados
   DWORD nNumBytesALer, // número de bytes a ler
   LPDWORD lpNumBytesLidos,  // endereço do número de bytes lidos
   LPOVERLAPPED lpSobreposta // endereço da estrutura para os dados
);
```

A função *ReadFile* aceita parâmetros idênticos aos da função *WriteFile* que a dica anterior detalhou, exceto que a função *ReadFile* espera o parâmetro *nNumBytesALer* em vez do parâmetro *nNumBytesAGravar*, e retorna o parâmetro *lpNumBytesLidos* em vez de *lpNumBytesGravados*. O parâmetro *nNumBytesALer* especifica o número de bytes que *ReadFile* deve ler do arquivo. O parâmetro *lpNumBytesLidos* aponta para o número de bytes lidos. *ReadFile* define esse valor como zero antes de fazer qualquer trabalho ou verifição de erro. Se o parâmetro *lpNumBytesLidos* for zero quando *ReadFile* retornar *True* em uma canalização nomeada, a outra ponta da canalização no modo de mensagem chamará a função *WriteFile* com *nNumBytesAGravar* definida com o valor 0.

Se *lpSobreposta* for *NULL*, *lpNumBytesLidos* não poderá ser *NULL*. Se *lpSobreposta* não for *NULL*, *lpNumBytesLidos* poderá ser *NULL*. Se essa for uma operação de leitura sobreposta, você poderá chamar *GetOverlappedResult* para obter o número de bytes lidos. Se *hArquivo* estiver associado com uma porta de E/S, você poderá chamar *GetQueuedCompletionStatus* para obter o número de bytes lidos.

Se *hArquivo* foi aberto com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, a operação de leitura iniciará no deslocamento que você especificou na estrutura *OVERLAPPED*, e *ReadFile* poderá retornar antes que a operação de leitura tenha sido completada. Nesse caso, *ReadFile* retorna *False*, e a função *GetLastError* retorna *ERROR_IO_PENDING*. Isso permite que o processo chamador continue enquanto a operação de leitura termina. O sistema operacional define o evento que você especificou na estrutura *OVERLAPPED* ao estado sinalizado ao completar a operação de leitura.

Se *hArquivo* não foi aberto com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* for *NULL*, a operação de leitura iniciará na posição de arquivo atual, e *ReadFile* não retornará até que a operação termine. Se o processo não abriu *hArquivo* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, a operação de leitura iniciará no deslocamento que você especificou na estrutura *OVERLAPPED*. *ReadFile* não retornará até que o sistema operacional complete a operação de leitura.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função retornar um valor diferente de zero e o número de bytes lidos for zero, o ponteiro de arquivo estará além da ponta final do arquivo ao tempo da operação de leitura. No entanto, se o processo abriu o arquivo com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, o valor de retorno da função será *False*, e *GetLastError* retornará *ERROR_HANDLE_EOF* quando o ponteiro de arquivo estiver além do final atual do arquivo. Se a função falhar, o valor de retorno da função será zero.

ReadFile retornará quando um dos casos seguintes for verdadeiro: uma operação de escrita completa no ponto de escrita da canalização, o número de bytes solicitado tiver sido lido ou um erro ocorrer. Caso outro processo bloqueie parte do arquivo e a operação de leitura sobreponha a parte bloqueada, a função *ReadFile* falhará. Os aplicativos não deverão ler ou gravar no buffer de entrada que uma operação de leitura estiver usando até que a operação de leitura termine. Um acesso prematuro ao buffer de entrada poderá levar à corrupção dos dados lidos nesse buffer. Os caracteres podem ser lidos a partir do buffer de entrada do console usando *ReadFile* com um indicativo para a entrada no console. O modo de console determina o comportamento exato da função *ReadFile*.

Se o processo estiver lendo uma canalização nomeada no modo de mensagem e a próxima mensagem for maior do que o parâmetro *nNumBytesALer* especificar, *ReadFile* retornará *False*, e *GetLastError* retornará *ERROR_MORE_DATA*. Uma chamada subseqüente à função *ReadFile* ou *PeekNamedPipe* pode ler o restante da mensagem. Quando você lê a partir de um dispositivo de comunicações, os tempos-limite de comunicação atuais (como definidos e lidos usando-se as funções *SetCommTimeouts* e *GetCommTimeouts*) governam o comportamento da função *ReadFile*. Poderão ocorrer resultados imprevisíveis se você deixar de definir os valores de tempo-limite. Se *ReadFile* tentar ler a partir de um mailslot cujo buffer é pequeno demais, a função retornará *False*, e *GetLastError* retornará *ERROR_INSUFFICIENT_BUFFER*.

Se um processo fechou o indicativo de canalização de escrita anônima, e *ReadFile* tentar usar o indicativo de canalização de leitura anônima para leitura, a função retornará *False*, e *GetLastError* retornará *ERROR_BROKEN_PIPE*. A função *ReadFile* poderá falhar e retornará *ERROR_INVALID_USER_BUFFER* ou *ER-*

ROR_NOT_ENOUGH_MEMORY sempre que existirem muitas solicitações de E/S assíncronas não-atendidas. O programa *Write_File.cpp* que a dica anterior apresentou mostra o uso da função *ReadFile*.

*Nota: O código **ReadFile** para conferir a condição de fim de arquivo (eof) difere para as operações síncronas e assíncronas. Quando uma operação de leitura síncrona atinge o final de um arquivo, **ReadFile** retorna **True** e define ***lpNumBytesLidos** como 0. Quando uma operação de leitura assíncrona chega ao final de um arquivo, **ReadFile** falha e retorna **ERROR_HANDLE_EOF**.*

1458 FECHANDO O ARQUIVO

Você, usa indicativos de arquivo para trabalhar com arquivos a partir de dentro de seus programas Windows. Como com outros indicativos (tais como indicativo para memória ou um indicativo para um dispositivo do contexto), você deverá sempre fechar um indicativo de arquivo após seu programa completar seu processamento. A função *CloseHandle* fecha um indicativo de objeto aberto. Você usará a função *CloseHandle* dentro de seus programas, como segue:

```
BOOL CloseHandle(HANDLE hObjeto);
```

O parâmetro *hObjeto* identifica um indicativo de objeto aberto. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará zero. Você pode usar a função *CloseHandle* para fechar indicativos para os seguintes objetos:

- Entrada e saída no console
- Arquivo de evento
- Mapeamento de arquivo
- Mutex
- Canalização nomeada
- Processo
- Semáforo
- Encadeamento
- Token (Windows NT somente)

A função *CloseHandle* invalida o indicativo de objeto especificado, decrementa o contador de indicativo do objeto, e efetua verificações de retenção de objeto. Após o processo fechar o último indicativo para um objeto, o sistema operacional remove o objeto do processamento. A função *CloseHandle* não fecha objetos de módulo. Fechar um indicativo inválido gera uma exceção. Isso inclui fechar um indicativo duas vezes, não verificando o valor de retorno e fechando um indicativo inválido e tentando usar *CloseHandle* em um indicativo que *FindFirstFile* retorna.

*Nota: Use **CloseHandle** para fechar indicativos que a função **CreateFile** retorna. Use **FindClose** para fechar indicativos que a função **FindFirstFile** retorna. Você aprenderá mais sobre **FindClose** na Dica 1476, mais à frente.*

1459 COMPARTILHANDO DADOS COM O MAPEAMENTO DE ARQUIVOS

Mapeamento de arquivos é a cópia do conteúdo de um arquivo ao espaço de endereço virtual de um processo. Quando você mapear um arquivo, a cópia do conteúdo do arquivo será conhecida como *exibição do arquivo*, e a estrutura interna que o seu programa usa para manter a cópia será conhecida como o *objeto de mapeamento de arquivo*. Para compartilhar dados, outro processo pode usar o objeto de mapeamento de arquivo do primeiro processo para criar uma exibição de arquivo idêntica ao seu próprio espaço de endereço virtual.

Um exemplo comum de processos que compartilham dados é o intercâmbio dinâmico de dados (DDE) e sua "irmã mais nova", a incorporação e ligação de objetos (OLE). No Windows 3.x, os aplicativos alocam memória global com o sinalizador *GMEM_DDESHARE*, e usam o indicativo de memória para compartilhar dados entre os processos. No Windows 95 e no Windows NT, os aplicativos devem, em vez disso, usar o mapeamento de arquivos. Não é necessário para seus aplicativos realmente ter um arquivo para mapear na memória. Um aplicativo poderá especificar um valor *0xFFFFFFFF* para o indicativo de arquivo quando ele chamar a função *CreateFileMapping*, que então mapeará uma exibição do arquivo de paginação do sistema operacional (sobre o que você aprendeu anteriormente) na memória. A dica a seguir explicará a função *CreateFileMapping* em detalhes.

Quando você mapear um arquivo na memória, seus programas podem essencialmente acessar o conteúdo do arquivo como se o arquivo fosse uma matriz — seus programas podem até usar valores de índice e ponteiros para acessar o conteúdo do arquivo.

Mapeando um Arquivo com a Memória Virtual 1460

Como você aprendeu na dica anterior, seus programas freqüentemente mapeiam os arquivos no espaço do endereço virtual de um processo. Mapear arquivos ajudará seus programas a acessar os dados dentro dos arquivos mais rapidamente e eficientemente. Quando você quiser mapear um arquivo na memória, seus programas deverão usar a função *CreateFileMapping* para criar um objeto de mapeamento de arquivo nomeado ou não-nomeado para o arquivo que você especificar. Você usará a função *CreateFileMapping* com seus programas, como segue:

```
HANDLE CreateFileMapping(
   HANDLE hArquivo,           // indicativo para o arquivo a
                              // mapear
   LPSECURITY_ATTRIBUTES lpAtribArqMapeamento,
                              // atributos de segurança
                              // opcionais
   DWORD flProtege,           // proteção para o objeto de
                              // mapeamento
   DWORD dwTamMaxMais,        // tamanho do objeto, 32 bits
                              // mais significativos
   DWORD dwTamMaxMenos,       // tamanho do objeto, 32 bits
                              // menos significativos
   LPCTSTR lpNome             // nome do objeto de mapeamento
                              // de arquivo
);
```

A função *CreateFileMapping* aceita os parâmetros mostrados na Tabela 1460.1.

Tabela 1460.1 Os parâmetros para a função ***CreateFileMapping***.

Parâmetro	Descrição
hArquivo	Identifica o arquivo a partir do qual criar um objeto de mapeamento. Você precisa abrir o arquivo com um modo de acesso compatível com os sinalizadores de proteção que o parâmetro *flProtege* especifica. A Microsoft recomenda que você abra os arquivos que pretende mapear para o acesso exclusivo. Se *hArquivo* for *(HANDLE)0xFFFFFFFF*, o processo chamador também precisará especificar o tamanho de um objeto de mapeamento nos parâmetros *dwTamMaxMais* e *dwTamMaxMin*. A função cria um objeto de mapeamento de arquivo do tamanho que o processo chamador especifica que o arquivo de paginação do sistema operacional utiliza, em vez de um arquivo nomeado no sistema de arquivos. Os processos podem compartilhar o objeto de mapeamento de arquivos por meio da duplicação, por meio da herança, ou por nome.
lpAtribMapeamentoArq	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribMapeamentoArq* for *NULL*, os processos-filho não poderão herdar o indicativo.
flProtege	Especifica a proteção que você quer para a exibição do arquivo, quando ele for mapeado: *PAGE_READONLY*, que dá acesso de leitura somente para a região comprometida de páginas. Uma tentativa de gravar ou executar a região comprometida resulta em uma violação de acesso. O arquivo que o parâmetro *hArquivo* especifica precisa ter sido criado com o acesso *GENERIC_READ*. *PAGE_READWRITE*, que dá acesso de leitura e gravação à região comprometida de páginas. O arquivo que *hArquivo* especifica precisa ter sido criado com o acesso *GENERIC_READ* e *GENERIC_WRITE*.

Tabela 1460.1 Os parâmetros para a função ***CreateFileMapping***. (*Continuação*)

Parâmetro	Descrição
	PAGE_WRITECOPY, que dá cópia ao acesso de escrita para a região de páginas comprometidas. Os arquivos que *hArquivo* especifica precisam ter sido criados com o acesso *GENERIC_READ* e *GENERIC_WRITE*. Além disso, um aplicativo pode combinar (usando o operador *OU* bit a bit) um ou mais dos valores de atributo da seção listados na Tabela 1460.2 com um dos três valores precedentes de proteção de página para especificar certos atributos de seção.
dwMaxTamMais	Especifica os 32 bits mais significativos do tamanho máximo do objeto de mapeamento de arquivo.
dwMaxTamMenos	Especifica os 32 bits menos significativos do tamanho máximo do objeto de mapeamento de arquivo. Se este parâmetro e *dwMaxTamMais* forem zero, o tamanho máximo do objeto de mapeamento de arquivo será igual ao tamanho atual do arquivo que *LArquivo* identifica.
lpNome	Aponta para uma string terminada por *NULL* que especifica o nome do objeto de mapeamento. O nome pode conter qualquer caractere exceto a barra invertida (\). Se esse parâmetro for igual ao nome de um objeto de mapeamento nomeado existente, a função solicitará acesso ao objeto de mapeamento com a proteção que *flProtege* especifica. Se esse parâmetro for *NULL*, a função criará o objeto de mapeamento sem um nome.

Como você aprendeu na Tabela 1460.1, é possível combinar os valores de proteção da página para o arquivo com um ou mais valores de atributo de seção, como mostrado na Tabela 1460.2.

Tabela 1460.2 Valores de proteção de página para o arquivo mapeado na memória.

Valor	Descrição
SEC_COMMIT	Aloca espaço físico na memória ou no arquivo de paginação no disco para todas as páginas de uma seção. Essa é a definição padrão.
SEC_IMAGE	O arquivo que você especificou para o mapeamento de arquivo da seção em um arquivo de imagem executável. Como as informações de mapeamento e a proteção de arquivo são tiradas do arquivo de imagem, nenhum outro atributo é válido com *SEC_IMAGE*.
SEC_NOCACHE	A função deverá definir todas as páginas de uma seção com *não-cacheáveis*. Em máquinas 80x86 e MIPS, usar o cache para essas estruturas somente reduz o desempenho — já que o hardware mantém os caches coerentes. Alguns controladores de dispositivo requerem dados que não estão no cache para que os programas possam escrever na memória física. *SEC_NOCACHE* requer que *SEC_RESERVE* ou *SEC_COMMIT* também esteja definida.
SEC_RESERVE	Reserva todas as páginas de uma seção sem alocar armazenagem física. Nenhuma outra operação de alocação pode usar o intervalo reservado de página até que ele esteja liberado. O aplicativo pode comprometer páginas reservadas em chamadas subseqüentes à função *VirtualAlloc*. Esse atributo será válido somente se o parâmetro *hArquivo* for *(HANDLE)0xFFFFFFFF*, isto é, um objeto de mapeamento de arquivo que o arquivo de paginação que o sistema operacional oferece

Se a função for bem-sucedida, ela retornará um indicativo para o objeto de mapeamento de arquivo. Se o objeto de mapeamento existiu antes da chamada da função, a função *GetLastError* retorna *ERROR_ALREADY_EXISTS*, e a função retorna um indicativo válido para o objeto de mapeamento de arquivo existente (com seu tamanho atual, não o novo tamanho especificado). Se o objeto de mapeamento não existia, *GetLastError* retorna zero. Se a função falhar, ela retornará *NULL*.

Após um processo ter criado um objeto de mapeamento de arquivo, o tamanho do arquivo não deverá exceder o tamanho do objeto de mapeamento de arquivo; se exceder, nem todo o conteúdo do arquivo estará dis-

ponível para os aplicativos compartilharem. Se um aplicativo especifica um tamanho para o objeto de mapeamento de arquivo que é maior que o tamanho do arquivo nomeado atual no disco, o Windows cresce o arquivo no disco para ficar igual ao tamanho especificado do objeto de mapeamento de arquivo. O indicativo que *CreateFileMapping* retorna tem total acesso ao novo objeto de mapeamento de arquivo. O aplicativo pode usar o indicativo com qualquer função que requeira um indicativo para um objeto de mapeamento de arquivo. Os processos podem compartilhar objetos de mapeamento de arquivo por meio da criação de processo, por meio da duplicação do indicativo, ou por nome.

Nota: Sob o Windows 95, você não deve usar indicativos de arquivo que usou para criar objetos de mapeamento de arquivo em chamadas subseqüentes às funções de E/S de arquivo, tais como ReadFile e WriteFile. Em geral, se você usou um indicativo de arquivo em uma chamada bem-sucedida para a função CreateFileMapping, não use esse indicativo a não ser que primeiro feche o objeto de mapeamento de arquivo correspondente.

Criar um objeto de mapeamento de arquivo cria o potencial para mapear uma exibição do arquivo mas não mapeia a exibição. As funções *MapViewOfFile* e *MapViewOfFileEx* mapeiam uma exibição de um arquivo no espaço de endereço de um processo. A Dica 1461 explica a função *MapViewOfFile* em detalhes.

Com uma exceção importante, as exibições de arquivo que derivaram de um único objeto de mapeamento de arquivo são coerentes, ou idênticas, em um determinado tempo. Se múltiplos processos tiverem indicativos do mesmo objeto de mapeamento de arquivo, eles verão uma exibição coerente dos dados ao mapearem uma exibição do arquivo. A exceção tem a ver com arquivos remotos. Embora *CreateFileMapping* trabalhe com arquivos remotos, ela não os mantém coerentes. Por exemplo, se dois computadores mapearem um arquivo como gravável, e ambos modificarem a mesma página, cada computador verá somente suas próprias escritas na página. Quando os dados forem atualizados no disco, o sistema operacional não mesclará os dados. Além disso, um arquivo mapeado e um arquivo que as funções de entrada e saída (*ReadFile* e *WriteFile*) acessam não são necessariamente coerentes.

Para fechar totalmente um objeto de mapeamento de arquivo, um aplicativo precisa chamar *UnmapViewOfFile* para desmapear todas as exibições mapeadas do objeto, e chamar *CloseHandle* para fechar o indicativo do objeto de mapeamento de arquivo. A ordem em que um aplicativo chama essas funções não importa. A chamada à *UnampViewOfFile* é necessária porque as exibições mapeadas de um objeto de mapeamento de arquivo mantêm indicativos de abertura internos para o objeto, e um objeto de mapeamento de arquivo não fechará até que o processo feche todos os indicativos abertos para o objeto de mapeamento de arquivo.

MAPEANDO UMA EXIBIÇÃO DE UM ARQUIVO NO PROCESSO ATUAL 1461

Após seus programas criarem um objeto de mapeamento de arquivo, eles precisam então mapear o arquivo no objeto. Para fazer isso, seus programas usarão a função *MapViewOfFile*. Essa função mapeia uma exibição de um arquivo no espaço de endereço do processo chamador. Você usará a função *MapViewOfFile* dentro de seus programas, como mostrado no protótipo a seguir:

```
LPVOID MapViewOfFile(
   HANDLE  hObjMapeamentoArq, // objeto de mapeamento de
                              // arquivo para mapear no
                              // espaço de endereço
   DWORD dwAcessoDesejado, // modo de acesso
   DWORD dwDeslocArqMais,  // 32 bits mais significativos
                              // do deslocamento do arquivo
   DWORD dwDeslocArqMenos, // 32 bits menos significativos do deslocamento
                              // do arquivo
   DWORD dwNumBytesMapear // número de bytes a mapear
);
```

O parâmetro *hObjMapeamentoArq* identifica um indicativo aberto de um objeto de mapeamento de arquivo. As funções *CreateFileMapping* e *OpenFileMapping* retornam esse indicativo. O parâmetro *dwAcessoDesejado* especifica o tipo de acesso à exibição do arquivo e, portanto, a proteção das páginas que o arquivo mapeia. Esse parâmetro pode ser um dos valores que a Tabela 1461 lista.

Tabela 1461 Valores possíveis para o parâmetro *dwAcessoDesejado*.

Valor	Significado
FILE_MAP_WRITE	Acesso de leitura e escrita O parâmetro *hObjMapeamentoArq* precisa ter sido criado com a proteção *PAGE_READWRITE*. O sistema operacional mapeia uma exibição de leitura e gravação do arquivo.
FILE_MAP_READ	Acesso de leitura somente O parâmetro *hObjMapeamentoArq* precisa ter sido criado com a proteção *PAGE_READWRITE* ou *PAGE_READONLY* O sistema operacional mapeia uma exibição de leitura somente do arquivo
FILE_MAP_ALL_ACCESS	O mesmo que *FILE_MAP_WRITE*.
FILE_MAP_COPY	Cópia no acesso de escrita. Caso crie o mapa com *PAGE_WRITECOPY* e a exibição com *FILE_MAP_COPY*, você receberá uma exibição para arquivo. Se você escrever nela, as páginas serão automaticamente permutáveis, e as modificações que você fizer não irão para o arquivo de dados original Sob o Windows 95, você precisa passar PAGE_WRITECOPY para *CreateFileMapping*; caso contrário, o Windows retornará um erro. Sob o Windows NT, não há restrição sobre como você precisa criar o parâmetro *hObjMapeamentoArq*. Copiar na escrita é válido para qualquer tipo de exibição. Se você usar *DuplicateHandle* ou *OpenFileMapping* para compartilhar o mapeamento entre múltiplos processos, e um processo escrever em uma exibição, a modificação será propagada para o outro processo. O arquivo original não muda.

O parâmetro *dwDeslocArqMais* especifica os 32 bits mais significativos do deslocamento de arquivo onde o mapeamento deve iniciar. O parâmetro *dwDeslocArqMenos* especifica os 32 bits menos significativos do deslocamento do arquivo onde o mapeamento deve começar. A combinação de deslocamentos mais e menos significativos precisa especificar um deslocamento dentro do arquivo que é igual à granularidade da alocação de memória do sistema, ou a função falha. Isto é, o deslocamento precisa ser um múltiplo da granularidade da alocação (tal como 8 ou 16 bytes). O parâmetro *dwNumBytesAMapear* especifica o número de bytes do arquivo a mapear. Se *dwNumBytesAMapear* for zero, o sistema operacional mapeará o arquivo inteiro.

Se a função for bem-sucedida, ela retornará o endereço inicial da exibição mapeada. Se a função falhar, ela retornará *NULL*. Mapear um arquivo torna a porção especificada do arquivo visível no espaço de endereço do processo chamador.

O CD-ROM que acompanha este livro inclui o programa *Sample_Map.cpp*. Este programa cria um arquivo mapeado na memória quando o aplicativo inicia. Quando o usuário seleciona a opção *Testar!*, o programa mapeia uma exibição para o arquivo e coloca os dados na memória. Em seguida, o programa cria um encadeamento e aguarda um temporizador para que o encadeamento modifique os dados. O encadeamento usa os dados para exibir uma caixa de mensagem e coloca a string "Received" na memória mapeada quando o usuário fecha a caixa de mensagem. Quando o programa recebe uma mensagem *WM_TIMER*, ele verifica se o encadeamento colocou a string na memória mapeada. Se tiver, o programa terminará o encadeamento, e o temporizador alertará o usuário.

1462 Abrindo um Objeto de Mapeamento de Arquivo Nomeado

Seus programas usarão a função *CreateFileMapping* para criar um mapa de arquivo nomeado, ou não-nomeado, e a função *MapViewOfFile* para mapear o arquivo na memória. No entanto, no programa *Sample_Map.cpp*, um segundo encadeamento abre o arquivo mapeado. Em vez de criar um novo mapeamento de arquivo, o segundo encadeamento usa a função *OpenFileMapping* para abrir o arquivo mapeado para seu próprio uso Você usará a função *OpenFileMapping* dentro de seus programas, como mostrado no seguinte protótipo:

```
HANDLE OpenFileMapping(DWORD dwAcessoDesejado,
                       BOOL bIndicHeranca, LPCTSTR lpNome);
```

O parâmetro *dwAcessoDesejado* especifica o acesso para o objeto de mapeamento de arquivo. Sob o Windows NT, o sistema operacional verifica o parâmetro de acesso com relação a qualquer descritor de segurança no objeto de mapeamento de arquivo-alvo. Esse parâmetro pode ser um dos valores já listados anteriormente na Tabela 1461. O parâmetro *bIndicHeranca* especifica se um novo processo deve herdar o indicativo retornado du-

rante a criação do processo. Um valor *True* indica que o novo processo herda o indicativo. O parâmetro *lpNome* aponta para uma string que nomeia o objeto de mapeamento de arquivo que o processo deverá abrir. Se houver um indicativo aberto a um objeto de mapeamento de arquivo por esse nome, e o descritor de segurança no objeto de mapeamento não entrar em conflito com o parâmetro *dwAcessoDesejado*, a operação de abertura será bem-sucedida.

Se a função for bem-sucedida, ela retornará um indicativo de abertura ao objeto de mapeamento de arquivo especificado. Se a função falhar, ela retornará *NULL*. Você pode usar o indicativo *OpenFileMapping* com qualquer função que requeira um indicativo a um objeto de mapeamento de arquivo.

COMPREENDENDO OS ATRIBUTOS DE ARQUIVO 1463

Na seção Arquivos, Diretórios e Discos, você aprendeu sobre os atributos de arquivo que o DOS associa com cada arquivo que você cria. Similarmente, o Windows associa um conjunto de atributos com todo arquivo. O Windows inicia muitos dos atributos de arquivo quando você cria o arquivo, e, mais tarde, altera alguns atributos toda vez que você acessa o arquivo. Mais frequentemente, você não irá querer alterar os atributos de um arquivo, mas simplesmente lê-los e reagir apropriadamente. Similar ao DOS, a maioria dos atributos de arquivo tem a ver com as definições de sinalizadores, tamanho do arquivo e a data e a hora dos arquivos. Em dicas subseqüentes, você manipulará e acessará alguns dos atributos de arquivo mais comumente usados.

OBTENDO E ALTERANDO OS ATRIBUTOS DE UM ARQUIVO 1464

Como visto na dica anterior, o Windows associa atributos de arquivo com cada arquivo que você cria dentro do sistema operacional. Exatamente como você lê atributos de arquivo com as funções padrão de C na seção Arquivos, Diretórios e Discos, a API do Windows fornece muitas funções que você pode usar para ler os atributos de um arquivo. A função *GetFileAttributes* retorna informações a partir de um subconjunto de todos os atributos possíveis para um arquivo ou diretório especificado. Dicas posteriores discutirão outras funções que retornam outros atributos de arquivo, tais como a hora ou o tamanho do arquivo. Você usará a função *GetFileAttributes* dentro de seus programas, como segue:

```
DWORD GetFileAttributes(LPCTSTR lpNomeArq);
```

O parâmetro *lpNomeArq* aponta para uma string terminada por *NULL* que especifica o nome de um arquivo ou de um diretório. Sob o Windows NT, há um tamanho de string padrão para caminhos de *MAX_PATH* caracteres. Esse limite resulta de como a função *GetFileAttributes* analisa os caminhos. Um aplicativo pode chamar a versão larga (W) de *GetFileAttributes* e inserir "\\?\" ao caminho para elevar esse limite e enviar em caminhos maiores que *MAX_PATH* caracteres. O "\\?\" informa a função para desativar a análise do caminho; ele permite que seus programas usem caminhos maiores que *MAX_PATH* com *GetFileAttributesW*. A função *GetFileAttributesW* também trabalha com nomes UNC. O sistema operacional ignora o "\\?\", como parte do caminho. Por exemplo, a função vê "\\?\C:\meusdocumentos\particulares" como "C:\meusdocumentos\particulares", e "\\?\jamsa[1]\happy\comida" como "\\jamsa\happy\comida". Sob o Windows 95, a string *lpNomeArq* não pode exceder *MAX_PATH* caracteres. O Windows 95 não suporta o prefixo "\\?\".

Se a função for bem-sucedida, o valor de retorno conterá os atributos do arquivo ou diretório especificado. Se a função falhar, ela retornará *0xFFFFFFFF*. Para obter as informações de erro estendidas, chame *GetLastError*.

Os atributos podem ser um ou mais dos valores listados na Tabela 1464.

Tabela 1464 Os valores de retorno da função ***GetFileAttributes***

Valor	Significado
FILE_ATTRIBUTE_ARCHIVE	O arquivo ou diretório deve ser incluído em uma operação de backup. Esse bit é usado somente pelos aplicativos de backup ou de remoção.
FILE_ATTRIBUTE_COMPRESSED	O arquivo ou diretório está compactado. Para um arquivo, isso significa que todos os dados no arquivo estão compactados. Para um diretório, significa que a compactação é o padrão para arquivos ou subdiretórios recém-criados.

Tabela 1464 Os valores de retorno da função ***GetFileAttributes***. (*Continuação*)

Valor	Significado
FILE_ATTRIBUTE_DIRECTORY	O objeto atual é um diretório.
FILE_ATTRIBUTE_HIDDEN	O arquivo ou diretório está oculto (não-incluído na listagem normal).
FILE_ATTRIBUTE_NORMAL	O arquivo ou diretório não tem outros atributos definidos. Este atributo somente é válido se usado sozinho
FILE_ATTRIBUTE_OFFLINE	Os dados do arquivo não estão disponíveis imediatamente. Indica que os dados do arquivo foram movidos fisicamente para o armazenamento off-line.
FILE ATTRIBUTE READONLY	O arquivo ou diretório é de leitura somente. Os aplicativos podem ler o arquivo mas não podem escrever nele ou excluí-lo. No caso de um diretório, os aplicativos não podem removê-lo.
FILE_ATTRIBUTE_SYSTEM	O arquivo ou diretório é parte do sistema operacional ou o sistema operacional usa o arquivo ou diretório exclusivamente.
FILE_ATTRIBUTE_TEMPORARY	Um processo está usando o arquivo para armazenagem temporária. Esse aplicativo deverá excluir um arquivo temporário assim que o aplicativo não precisar mais do arquivo.

Como você aprendeu, seus programas podem usar a função *GetFileAttributes* para obter os atributos de um arquivo. Ocasionalmente, no entanto, seus programas também podem modificar os atributos do arquivo. Você pode usar a função *SetFileAttributes* para definir os atributos de um arquivo, como mostrado no protótipo a seguir:

```
BOOL SetFileAttributes(
   LPCTSTR  lpNomeArq,      // endereço do nome de arquivo
   DWORD  dwAtribArquivo    // atributos a definir
);
```

O parâmetro *lpNomeArq* aponta para uma string que especifica o nome do arquivo cujos atributos a função deve definir. As mesmas limitações se aplicam ao parâmetro *lpNomeArq* conforme aplicado ao parâmetro *lpNomeArq* da função *GetFileAttributes*. O parâmetro *dwAtribArquivo* especifica os atributos de arquivo que a função deve definir. Este parâmetro pode ser uma combinação dos valores que a Tabela 1464 detalhou. No entanto, todos os outros valores anulam *FILE ATTRIBUTES_NORMAL*.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, ela retornará zero. Para obter informações estendidas de erro, chame *GetLastError*. Você não pode usar *SetFileAttributes* para definir o estado de compactação de um arquivo. Definir *FILE_ATTRIBUTE_COMPRESSED* no parâmetro *dwAtribArquivo* nao faz nada. Use a função *DeviceIoControl* e a operação *FSCTL_SET_COMPRESSION* para definir o estado de compactação de um arquivo.

O CD-ROM que acompanha este livro inclui o programa *Check_ReadOnly.cpp*. Este programa verifica o arquivo *file.dat* para determinar se o atributo de leitura somente está definido. Se estiver, o programa limpará todas as definições de atributos e depois excluirá o arquivo.

1465 Obtendo o Tamanho de um Arquivo

O Windows associa certos atributos a cada arquivo que salva no disco. Uma das solicitações mais comuns que seus programas farao é obter o tamanho do arquivo em bytes. A função *GetFileSize* obtém o tamanho, em bytes, do arquivo especificado. Você usará esta função, como segue:

```
DWORD GetFileSize(
   HANDLE hArquivo,        // indicativo do arquivo
   LPDWORD lpTamArqMais    // endereço da palavra mais
                           // significativa para o tamanho do arquivo
);
```

O parâmetro *hArquivo* especifica o indicativo aberto do arquivo cujo tamanho a função deve retornar. O processo precisa ter criado o indicativo com o acesso *GENERIC_READ* ou *GENERIC_WRITE* ao arquivo. O

parâmetro *lpTamArqMais* aponta para a variável onde a função retornará a palavra mais significativa do tamanho do arquivo. Caso o aplicativo não requeira a palavra mais significativa, o parâmetro *lpTamArqMais* poderá ser *NULL*.

Se a função for bem-sucedida, ela retornará a palavra dupla (*doubleword*) menos significativa do tamanho do arquivo, e, se *lpTamArqMais* não for *NULL*, ela colocará a doubleword mais significativa do tamanho do arquivo na variável para a qual o parâmetro aponta. Caso a função falhar e *lpTamArqMais* for *NULL*, a função retornará *0xFFFFFFFF*. Para obter informações estendidas de erro chame *GetLastError*. Se a função falhar e *lpTamArqMais* não for *NULL*, a função retornará *0xFFFFFFFF* e *GetLastError* retornará um valor diferente de *NO_ERROR*.

Você não pode usar a função *GetFileSize* com um indicativo de um dispositivo que não efetua busca, tal como uma canalização ou um dispositivo de comunicações. Para determinar o tipo de arquivo para *hArquivo*, use a função *GetFileType*. A função *GetFileSize* obtém o tamanho não-compactado do arquivo. Use a função *GetCompressedFileSize* para obter o tamanho de um arquivo compactado. Observe que, se a função retornar *0xFFFFFFFF* e *lpTamArqMais* não *for NULL*, um aplicativo precisará chamar *GetLastError* para determinar se a função foi bem-sucedida ou se falhou.

O CD-ROM que acompanha este livro inclui o programa *Check_FileSize.cpp*, que define uma função, *DisplayFileSize*, que primeiro usa *GetFileType* para verificar o tipo do indicativo de arquivo. Se o arquivo estiver baseado no disco, a funçao *DisplayFileSize* chamará *GetFileSize* para obter o tamanho do arquivo, e exibirá o tamanho dentro de uma caixa de mensagem. Se o arquivo não estiver baseado no disco, a função alertará o usuário de que o arquivo não estará baseado no disco e terminará.

Obtendo a Data e a Hora de um Arquivo 1466

Em dicas anteriores você aprendeu como verificar a data e a hora de um arquivo usando as funções da biblioteca em tempo de execução de C. A API do Windows também oferece uma função que seus programas podem usar para obter a data e a hora de um arquivo. A função *GetFileTime* recupera a data e a hora em que um arquivo foi criado, acessado ou modificado pela última vez. Você usará a função *GetFileTime* dentro de seus programas, como segue:

```
BOOL GetFileTime(
   HANDLE hArquivo,                  // identifica o arquivo
   LPFILETIME lpHoraCriacao,         // endereço da hora de criação
   LPFILETIME lpHoraUltimoAcesso,    // endereço da hora do último acesso
   LPFILETIME lpHoraUltimaGravacao // endereço da hora da última gravação
```

O parâmetro *hArquivo* identifica os arquivos para os quais a função deve obter as datas e as horas. O processo precisa ter criado o indicativo de arquivo com o acesso *GENERIC_READ* para o arquivo. O parâmetro *lpHoraCriacao* aponta para uma estrutura *FILETIME* para receber a data e a hora em que o arquivo foi criado. O parâmetro *lpHoraUltimoAcesso* aponta para uma estrutura *FILETIME* para receber a data e a hora em que o arquivo foi acessado pela última vez. A hora do último acesso inclui a hora em que o arquivo foi gravado ou lido pela última vez, ou, no caso de arquivos executáveis, executado. O parâmetro *lpHoraUltimaGravacao* aponta para uma estrutura *FILETIME* para receber a data e a hora em que o arquivo foi gravado pela última vez. Caso o arquivo não requeira informações sobre todos os três horários, os parâmetros para as vezes que a função não requis informações poderá ter um valor *NULL*.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará o valor zero. Para obter informações de erro estendidas, chame *GetLastError*.

Os sistemas de arquivo do Windows 95 e do Windows NT suportam a criação de arquivo, último acesso e últimos valores de hora de escrita. No Windows 95, a precisão para o tempo de arquivo é 2 segundos (o que significa que o horário que o Windows registra dentro do atributo tem uma diferença de 2 segundos do horário real de acesso). A precisão do horário para os arquivos em outros sistemas de arquivo, tais como aqueles conectados em uma rede, depende do sistema de arquivo remoto. O dispositivo remoto também pode limitar a precisão do horário.

O CD-ROM que acompanha este livro inclui o programa *Get_Time.cpp*, que usa *GetFileTime* para obter a hora atual do arquivo e depois converter essa hora em hora local. O programa depois converte a estrutura *FILETIME* do horário local em um formato mais reconhecível de data e hora. Finalmente, o programa exibe a data e a hora convertidas dentro de uma caixa de mensagem.

1467 CRIANDO DIRETÓRIOS

Na Dica 396, você aprendeu como usar uma função padrão da biblioteca de execução da linguagem C para criar um diretório dentro do DOS. Similarmente, seus programas Windows podem usar a função *CreateDirectory* da API Win32 para criar um novo diretório. Se o sistema de arquivos subjacente suportar segurança em arquivos e diretórios, a função aplicará o descritor de segurança especificado ao novo diretório. Lembre-se, cada processo terá seu próprio diretório atual, de modo que as chamadas à *CreateDirectory* não devem presumir que o diretório atual para o processo é uma constante. Você usará a função *CreateDirectory* dentro de seus programas conforme mostrado no seguinte protótipo:

```
BOOL CreateDirectory(
   LPCTSTR lpNomeCaminho,     // ponteiro para uma string
                              // de caminho de diretório
   LPSECURITY_ATTRIBUTES lpAtribSegur
                              // ponteiro para um descritor de segurança
);
```

O parâmetro *lpNomeCaminho* aponta para uma string terminada por *NULL* que especifica o caminho do diretório a ser criado. Há um limite no tamanho padrão da string para os caminhos de *MAX_PATH* caracteres. Esse limite se relaciona ao modo como a função *CreateDirectory* analisa os caminhos. Você pode exceder os limites de caminho sob o Windows NT conforme já detalhou a Dica 1464.

O parâmetro *lpAtribSegur* é um ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribSegur* for *NULL*, os processos-filho não poderão herdá-lo. Sob o Windows NT, o membro *lpAtribSegur* da estrutura especifica um descritor de segurança para o novo diretório. Se *lpAtribSegur* for *NULL*, o diretório receberá um descritor de segurança padrão. O sistema de arquivos-alvo precisa suportar a segurança em arquivos e diretórios para esse parâmetro ter um efeito. Sob o Windows 95, o sistema operacional ignora o membro *lpSecurityDescriptor* da estrutura.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, ela retornará o valor zero. Para obter as informações de erro estendidas, chame *GetLastError*.

O CD-ROM que acompanha este livro inclui o programa *Create_NewDir.cpp*. O programa usa a função *CreateDirectory* para criar um novo diretório chamado "New Directory" na unidade C do computador no qual o programa estiver sendo executado.

Nota: Alguns sistemas de arquivo, tais como o NT File System (NTFS), suportam a compactação para arquivos e diretórios individuais. Nas unidades formatadas para tal sistema de arquivos, um novo diretório herda o atributo de compactação de seu diretório-pai. Um aplicativo pode chamar CreateFile com o sinalizador FILE_FLAG_BACKUP_SEMANTICS ligado para obter um indicativo para um diretório. Para um exemplo de código, veja CreateFile.

1468 OBTENDO E DEFININDO O DIRETÓRIO ATUAL

Como visto na dica anterior, seus programas podem criar novos diretórios dentro do sistema de arquivo. No entanto, seus programas provavelmente requererão informações sobre o diretório atual mais freqüentemente do que criarão novos diretórios. A função *GetCurrentDirectory* lê o diretório atual para o processo atual. Você usará a função *GetCurrentDirectory* dentro de seus programa, como mostra o protótipo a seguir:

```
DWORD GetCurrentDirectory(
   DWORD nTamBuffer,   // tamanho, em caracteres do buffer
                       // do diretório
   LPTSTR lpBuffer,    // endereço do buffer para o
                       // diretório atual
);
```

O parâmetro *nTamBuffer* especifica o comprimento, em caracteres, do buffer para a string de diretório atual. O comprimento da string precisa incluir espaço para um caractere *NULL* finalizador. O parâmetro *lpBuffer* aponta para o buffer para a string do diretório atual Essa string terminada por *NULL* especifica o caminho absoluto para o diretório atual. Se a função for bem-sucedida, o valor de retorno especificará o número de caracteres escritos no buffer, não incluindo o caractere *NULL* finalizador. Se a função falhar, ela retornará zero. Para obter as informações de erro estendidas, chame *GetLastError*. Se o buffer apontado por *lpBuffer* não for grande o suficiente, o valor de retorno especificará o tamanho requerido para o buffer, incluindo o número de bytes necessários para um caractere *NULL* finalizador.

Similarmente, seus programas podem usar a função *SetCurrentDirectory* para mudar o diretório atual do processo atual para um diretório que você especificar. Você usará a função *SetCurrentDirectory* dentro de seus programas, como segue:

```
BOOL SetCurrentDirectory(LPCTSTR lpNomeCaminho);
```

O parâmetro *lpNomeCaminho* aponta para uma string terminada por *NULL* que especifica o caminho para o novo diretório atual. Esse parâmetro pode ser um caminho relativo ou um caminho totalmente qualificado do diretório especificado, e armazena esse caminho como o diretório atual. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará o valor zero. Para obter as informações estendidas de erro, chame *GetLastError*.

Cada processo tem um único diretório atual formado por duas partes.

- Um designador de disco que é uma letra de unidade seguida por dois-pontos, ou um nome de servidor e nome de compartilhamento (por exemplo, *nomeserv**nomecompart*).
- Um diretório no designador de disco.

O CD-ROM que acompanha este livro inclui o programa *Create_Set.cpp*. Quando você compilar e executar o programa, ele exibirá o diretório atual, depois criará o diretório *New Directory* e, finalmente, alterará o diretório atual para *New Directory* e exibirá o novo diretório atual.

OBTENDO OS DIRETÓRIOS WINDOWS E SYSTEM 1469

Você aprendeu como seus programas podem obter o diretório atual para o processo atual. À medida que seus programas se tornarem mais complexos, você frequentemente precisará de informações sobre a localização dos diretórios *Windows* e *System*. A função *GetWindowsDirectory* obtém o caminho do diretório Windows (que pode ser um diretório com um nome diferente de *c:\windows* ou *c:\winnt*). O diretório Windows contém arquivos como os aplicativos baseados no Windows, arquivos de inicialização e arquivos de Ajuda. Você usará a função *GetWindowsDirectory* dentro de seus programas, como mostra o protótipo a seguir:

```
UINT GetWindowsDirectory(
   LPTSTR lpBuffer,     // endereço do buffer para o diretório Windows
   UINT uTamanho        // tamanho do buffer do diretório
);
```

O parâmetro *lpBuffer* aponta para o buffer que receberá a string terminada por NULL que contém o caminho. Esse caminho não termina com uma barra invertida a não ser que o diretório Windows seja o diretório-raiz. Por exemplo, se o diretório chama-se *windows* na unidade C, o caminho do diretório Windows que essa função recuperou é *c:\windows*. Se o Windows foi instalado no diretório-raiz da unidade C, o caminho recuperado é C:\. O parâmetro *uTamanho* especifica o tamanho máximo, em caracteres, do buffer especificado pelo parâmetro *lpBuffer*. Você deve definir o parâmetro *uTamanho* com pelo menos *MAX_PATH* para permitir espaço suficiente no buffer para o caminho.

Se a função for bem-sucedida, ela retornará o comprimento, em caracteres, da string que a função copiou para o buffer, não incluindo o caractere *NULL* finalizador. Se o comprimento for maior que o tamanho do buffer, a função retornará o tamanho do buffer que requer para conter o caminho. Se a função falhar, ela retornará zero. Para obter informações estendidas de erro, chame *GetLastError*.

O diretório *Windows* é onde um aplicativo deve armazenar os arquivos de inicialização e de ajuda. Se o usuário estiver rodando uma versão compartilhada do Windows, o sistema operacional garantirá que o diretório *Windows* será privado para cada usuário. Se um aplicativo criar outros arquivos que ele quer armazenar de acordo com o usuário, deverá colocá-los no diretório especificado pela variável de ambiente *HOMEPATH*. *HOME-*

PATH sempre especifica o diretório normal do usuário, o que garantidamente é particular para cada usuário, ou o diretório default (por exemplo, *c:\users\default*) onde o usuário terá todo o acesso.

Assim como seus programas normalmente requererão informações sobre a localização do diretório Windows, eles também, com freqüência, requererão informações sobre o diretório system. A função *GetSystemDirectory* recupera o caminho do diretório System do Windows. Esse diretório contém arquivos como as bibliotecas, controladores e arquivos de fonte do Windows. Você usará a função *GetSystemDirectory* dentro de seus programas, como segue:

```
UINT GetSystemDirectory(
   LPTSTR  lpBuffer,    // endereço do buffer para o diretório system
   UINT uTamanho        // tamanho do buffer do diretório
);
```

Os parâmetros para *GetSystemDirectory* são os mesmos que para *GetWindowsDirectory*. Se a função *GetSystemDirectory* for bem-sucedida, a função retornará o comprimento, em caracteres, da string copiada para o buffer, não incluindo o caractere *NULL* finalizador. Se o comprimento for maior que o tamanho do buffer, a função retornará o tamanho do buffer que ela requer para armazenar o caminho. Se a função falhar, ela retornará zero. Para obter as informações estendidas de erro, chame *GetLastError*.

O CD-ROM que acompanha este livro inclui o programa *Show_Windows.cpp*. Quando você compilar e executar esse programa, ele usará as funções *GetWindowsDirectory* e *GetSystemDirectory* para obter os nomes dos diretórios Windows e System. O programa então exibirá os nomes de diretórios dentro da área cliente da janela.

Nota: Como regra, os aplicativos não devem criar arquivos no diretório System. Se o usuário estiver rodando uma versão compartilhada do Windows, o aplicativo não terá acesso de gravação no diretório System. Os aplicativos devem criar arquivos somente no diretório que a função GetWindowsDirectory retorne.

1470 REMOVENDO DIRETÓRIOS

Assim como seus programas podem criar diretórios temporários, ou diretórios que usará apenas internamente, algumas vezes seus programas precisarão remover um diretório existente. A função *RemoveDirectory* exclui um diretório existente vazio. Você usará a função *RemoveDirectory*, como mostrado no protótipo a seguir:

```
BOOL RemoveDirectory(LPCTSTR lpNomeCaminho);
```

O parâmetro *lpNomeCaminho* aponta para uma string terminada por *NULL* que especifica o caminho do diretório a ser removido. O caminho precisa especificar um diretório vazio, e o processo chamador precisa ter acesso de exclusão ao diretório. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará o valor zero. Para obter informações de erro estendidas, chame *GetLastError*.

O CD-ROM que acompanha este livro inclui o programa *Create_Remove.cpp*. Quando você compilar e executar esse programa, ele criará um novo diretório chamado *Child Folder*, e alertará o usuário de que fez isso. Em seguida, o programa excluirá o diretório recém-criado e também alertará o usuário dessa alteração.

1471 COPIANDO ARQUIVOS

Como você aprendeu, seus programas podem manipular arquivos dentro do Windows. À medida que seus programas se tornarem mais complexos, algumas vezes um programa precisará copiar um arquivo de um local para outro, mantendo o arquivo original em sua localização original. Você usará a função *CopyFile* dentro de seus programas, como mostrado aqui:

```
BOOL CopyFile(
   LPCTSTR lpNomeArqExist, // ponteiro para o nome do arquivo existente
   LPCTSTR lpNomeNovoArq,  // ponteiro para o nome de
                           // arquivo para onde copiar
   BOOL bFalhaSeExitir     // sinalizador para a operação
);                         // se o arquivo existir
```

O parâmetro *lpNomeArqExist* aponta para uma string terminada por *NULL* que especifica o nome de um arquivo existente. O parâmetro *lpNomeNovoArq* aponta para uma string terminada por *NULL* que especifica o nome do novo arquivo. O parâmetro *bFalhaSeExistir* especifica como a função deve proceder se um arquivo do mesmo nome que *lpNomeNovoArq* especifica já existir. Se o parâmetro *bFalhaSeExistir* for *True* e o novo arquivo já existir, a função falhará. Se esse parâmetro for *False* e o novo arquivo já existir, a função sobrescreverá o arquivo existente como bem-sucedido.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará zero. Para obter informações estendidas de erro, chame *GetLastError*. Os atributos de segurança para o arquivo existente não são copiados para o novo arquivo.

Os atributos de arquivo (*FILE_ATTRIBUTE_**) para o arquivo existente são copiados para o novo arquivo. Por exemplo, se um arquivo existente tiver o atributo de arquivo *FILE_ATTRIBUTE_READONLY*, uma cópia criada por meio de uma chamada para *CopyFile* também terá o atributo de arquivo *FILE_ATTRIBUTE_READONLY*.

O CD-ROM que acompanha este livro inclui o programa *Create_Copy.cpp*. Quando você compilar e executar esse programa, ele criará o arquivo *file1.txt*. Quando o usuário selecionar a opção *Testar!*, o programa copiará o arquivo *file1.txt* para o arquivo *file2.txt*. Se o arquivo *file2.txt* já existir, o programa perguntará se o usuário deseja substituir o arquivo existente.

Movendo e Renomeando Arquivos 1472

Seus programas podem facilmente criar uma cópia de um arquivo e colocá-lo em outra localização. No entanto, seus programas freqüentemente irão querer mover um arquivo ou um diretório para outra localização sem manter a cópia original do arquivo. A função *MoveFile* renomeia um arquivo ou um diretório existente (incluindo todos os seus filhos). Você usará a função *MoveFile* dentro de seus programas, como mostrado no protótipo a seguir:

```
BOOL MoveFile(
   LPCTSTR lpNomeArqExist, // endereço do nome do arquivo existente
   LPCTSTR lpNomeNovoArq   // endereço do nome novo para o arquivo
);
```

O parâmetro *lpNomeArqExist* aponta para uma string terminada por *NULL* que nomeia um arquivo ou diretório existente. O parâmetro *lpNomeNovoArq* aponta para uma string terminada por *NULL* que especifica o novo nome de um arquivo ou diretório. O novo nome não pode existir ainda. Um novo arquivo pode estar em um diferente sistema de arquivo ou unidade. Um novo diretório precisa estar na mesma unidade. Se a função for bem sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará o valor zero. Para obter informações estendidas de erro, chame *GetLastError*.

A função *MoveFile* moverá (renomeará) um arquivo ou um diretório (incluindo todos seus filhos), dentro do mesmo diretório ou em diretórios diferentes. A única limitação na função *MoveFile* é que ela falhará em movimentações entre diretórios que estejam em unidades de disco diferentes.

O CD-ROM que acompanha este livro inclui o programa *Move_Folder.cpp*. Quando você compilar e executar esse programa, ele primeiro criará uma estrutura de diretório temporário. Quando o usuário selecionar a opção *Testar!*, o programa moverá um dos diretórios da estrutura de diretórios para o outro diretório.

Apagando Arquivos 1473

Na Dica 1470, você aprendeu como seus programas podem usar a função *RemoveDirectory* para remover um diretório do sistema de arquivos. No entanto, como observado na Dica 1470, o diretório precisa estar vazio antes de você chamar *RemoveDirectory* ou a função falhará. Para remover arquivos dentro de um diretório, seus programas podem usar a função *DeleteFile*. Você usará a função *DeleteFile* dentro de seus programas, como segue:

```
BOOL DeleteFile(LPCTSTR lpNomeArq);
```

O parâmetro *lpNomeArq* aponta para uma string terminada por *NULL* que especifica o arquivo que a função deverá apagar. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se a função falhar, ela retornará um valor zero. Para obter informações estendidas de erro, chame *GetLastError*.

Se um aplicativo tentar apagar um arquivo que não existe, a função *DeleteFile* falhará. Sob o Windows 95, a função *DeleteFile* apagará um arquivo mesmo se ele estiver atualmente aberto para E/S normal ou como um arquivo mapeado na memória. Para evitar resultados errados, feche os arquivos antes de tentar apagá-los. Sob o

Windows NT, a função *DeleteFile* falhará se um aplicativo tentar apagar um arquivo que esteja atualmente aberto para E/S normal ou como um arquivo mapeado na memória.

1474 USANDO FINDFIRSTFILE PARA LOCALIZAR ARQUIVOS

Nas Dicas 390 e 391, você aprendeu como usar as funções padrão de biblioteca de execução C para pesquisar um arquivo no caminho de comandos e no diretório atual. Dentro de seus programas Windows, você deve usar as funções *Find* para localizar os arquivos que atendem a um determinado critério. Existem duas funções *Find* que seus programas podem usar. A função *FindFirstFile* pesquisará em um diretório o arquivo cujo nome você especificar. *FindFirstFile* examina os nomes de subdiretórios bem como os nomes de arquivos. Você usará a função *FindFirstFile* dentro de seus programas, como mostra o seguinte protótipo:

```
HANDLE FindFirstFile(
   LPCSTR lpNomeArq,      // ponteiro para nome de arquivo a pesquisar
   LPWIN32_FIND_DATA  lpDadosArqLocaliz  // ponteiro para as
                        // informações retornadas
);
```

Tanto sob o Windows 95 quanto sob o Windows NT o parâmetro *lpFileName* aponta para uma string terminada por *NULL* que especifica um diretório ou caminho e nome de arquivo válidos, que pode conter caracteres-chave (* e ?). Sob o Windows 95, no entanto, essa string não deve exceder *MAX_PATH* caracteres.

O parâmetro *lpDadosArqLocaliz* aponta para a estrutura *WIN32_FIND_DATA* que recebe informações sobre o arquivo ou subdiretório localizado. Seus programas podem usar a estrutura em chamadas subseqüentes às funções *FindNextFile* ou *FindClose* para referenciar o arquivo ou subdiretório. A estrutura *WIN32_FIND_DATA* descreve um arquivo que a função *FindFirstFile*, *FindFirstFileEx* ou *FindNextFile* encontrou. A API Win32 define a estrutura *WIN32_FIND_DATA*, como mostrado aqui:

```
typedef struct _WIN32_FIND_DATA {
   DWORD dwFileAttributes;
   FILETIME ftCreationTime;
   FILETIME ftLastAccessTime;
   FILETIME ftLastWriteTime;
   DWORD    nFileSizeHigh;
   DWORD    nFileSizeLow;
   DWORD    dwReserved0;
   DWORD    dwReserved1;
   TCHAR    cFileName [ MAX_PATH ];
   TCHAR    cAlternateFileName [ 14 ];
 } WIN32_FIND_DATA;
```

A Tabela 1474 explica a estrutura *WIN_FIND_DATA* em detalhes.

Tabela 1474 Os componentes da estrutura ***FILE_FIND_DATA***.

Membros	Descrição
dwFileAttributes	Especifica os atributos de arquivo do arquivo encontrado. Este membro pode ser um ou mais dos valores que a Tabela 1464 lista.
ftCreationTime	Especifica uma estrutura *FILETIME* que contém a hora em que o arquivo foi criado. *FindFirstFile* e *FindNextFile* informam os horários dos arquivos no formato Coordinated Universal Time (UTC). Estas funções definem os membros *FILETIME* como zero se o sistema de arquivos que contém o arquivo não suporta este membro de horário. Você pode usar a função *FileTimeToLocalFileTime* para converter de UTC para horário local, e, depois, usar a função *FileTimeToSystemTime* para converter o horário local para uma estrutura *SYSTEMTIME* que contém membros individuais para o mês, dia, ano, nome do dia da semana, hora, minuto, segundo e milissegundo.

Tabela 1474 Os componentes da estrutura *FILE_FIND_DATA*. (*Continuação*)

Membros	Descrição
ftLastAccessTime	Especifica uma estrutura *FILETIME* que contém o horário em que o arquivo foi acessado pela última vez. O horário está no formato UTC; os membros *FILETIME* serão zero se o sistema de arquivos não suportar esse membro de horário.
ftLastWriteTime	Especifica uma estrutura *FILETIME* que contém o horário em que o arquivo foi gravado pela última vez. O horário está no formato UTC; os membros *FILETIME* serão zero se o sistema de arquivos não suportar esse membro de horário.
nFileSizeHigh	Especifica o valor *DWORD* mais significativo do tamanho do arquivo, em bytes. Este valor será zero a não ser que o tamanho do arquivo seja maior que *MAXWORD*. O tamanho do arquivo é igual a (*nFileSizeHigh* MAXDWORD*) + *nFileSizeLow*.
nFileSizeLow	Especifica o valor *DWORD* menos significativo do tamanho do arquivo, em bytes.
dwReserved0	Reservado para uso futuro.
dwReserved1	Reservado para uso futuro.
cFileName	Uma string terminada por *NULL* que é o nome do arquivo.
cAlternateFileName	Uma string terminada por *NULL* que é um nome alternativo para o arquivo. Esse nome está no formato clássico de 8.3 (*nomearq.ext*).

Se um arquivo tem um nome de arquivo longo, o nome completo aparece no campo *cNomeArq*, e a versão no formato 8.3 truncado do nome aparece no campo *cAlternateFileName*. Caso contrário, *cAlternateFileName* estará vazio. Como uma alternativa, você pode usar a função *GetShortPathName* para localizar a versão de formato 8.3 de um nome de arquivo.

Se a função for bem-sucedida, ela retornará um indicativo de pesquisa que você usará em uma chamada subseqüente a *FindNextFile* ou *FindClose*. Se a função falhar, ela retornará *INVALID_HANDLE_VALUE*. Para obter informações de erro estendidas, chame *GetLastError*.

A função *FindFirstFile* abre um indicativo de pesquisa e retorna informações sobre o primeiro arquivo cujo nome corresponde ao padrão especificado. Após o indicativo de pesquisa estar estabelecido, você pode usar a função *FindNextFile* para pesquisar outros arquivos que correspondam ao mesmo padrão. Quando o indicativo de pesquisa não for mais necessário, use a função *FindClose* para fechá-lo. Esta função pesquisa os arquivos por nome somente; você não pode usá-lo para pesquisas baseadas em atributo. Para aprender sobre as pesquisas baseadas em atributo, veja a Dica 1477, mais à frente.

USANDO FINDNEXTFILE 1475

Na dica anterior, você aprendeu sobre a função *FindFirstFile*, que localiza a primeira ocorrência de um arquivo dentro de uma árvore de diretório que corresponde a um determinado nome de arquivo. No entanto, para continuar a pesquisar arquivos adicionais com o mesmo nome de arquivo (que corresponda ao caractere-chave), seus programas precisam usar a segunda função *Find*. A função *FindNextFile* continua uma pesquisa de arquivo a partir de uma chamada anterior à função *FindFirstFile*. Você usará a função *FindNextFile* dentro de seus programas conforme mostrado no protótipo a seguir:

```
BOOL FindNextFile(
   HANDLE  hLocalizArq,                    // indicativo a pesquisar
   LPWIN32_FIND_DATA  lpDadosArqLocaliz  // ponteiro para
                  // estrutura para o arquivo localizado
);
```

O parâmetro *hArqLocaliz* identifica um indicativo de pesquisa que uma chamada anterior à função *FindFirstFile* retorna. O parâmetro *lpDadosArqLocaliz* aponta para a estrutura *WIN32_FIND_DATA* que recebe informações sobre o arquivo ou subdiretório encontrado. Você pode usar a estrutura em chamadas subseqüentes à *FindNextFile* para referenciar o arquivo ou diretório encontrado.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, a função retornará um valor zero. Para obter informações estendidas de erro, chame *GetLastError*. Se *GetLastError* não encontrar arqui-

vos coincidentes, a função retornará *ERROR_NO_MORE_FILES*. A função *FindNextFile* pesquisa arquivos por nome somente; você não pode usá-la para pesquisas baseadas em atributo. Você aprenderá sobre as pesquisas baseadas em atributos na Dica 1477, logo à frente.

O CD-ROM que acompanha este livro inclui o programa *Walk_Directories.cpp*. Este programa usa *FindFirstFile* e *FindNextFile* para pesquisar a unidade de disco que você especificar. Adicionalmente, o programa usa uma chamada de função recursiva para garantir que a função pesquise todos os diretórios na unidade atual. À medida que o programa pesquisa, ele acrescenta nomes de arquivo na caixa de lista exibida.

1476 Fechando o Indicativo de Pesquisa com FindClose

Quando seus programas trabalham com as funções *FindFirstFile* e *FindNextFile*, você abre um indicativo diferente (um indicativo de pesquisa) para essas funções que o sistema operacional normalmente retornaria com uma chamada à *CreateFile*. Se você tentar fechar um indicativo de pesquisa com *CloseHandle*, seus programas retornarão um erro. Em vez disso, seus programas precisam usar a função *FindClose* para fechar o indicativo de pesquisa. Você usará a função *FindClose* dentro de seus programas, como mostra o protótipo a seguir:

```
BOOL FindClose(HANDLE hLocalizArq);
```

O parâmetro *hLocalizArq* identifica o indicativo de pesquisa. Seu programa precisa ter aberto anteriormente o indicativo de pesquisa com uma chamada à função *FindFirstFile*. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, a função retornará zero. Após o programa chamar a função *FindClose*, ele não pode usar o indicativo de pesquisa que o parâmetro *hLocalizArq* especifica em chamadas subseqüentes à *FindNextFile* ou *FindClose*. Dentro do programa *Walk_Directories.cpp*, detalhado anteriormente na Dica 1475, o programa usará a função *FindClose* apenas quando o usuário terminar de percorrer todos os diretórios de arquivo.

1477 Pesquisando por Atributos com as Funções FindFile

Como você aprendeu na Dica 1475, seus programas podem usar uma função de pesquisa recursiva (*WalkDirsRecurse* dentro do programa *Walk_Directories*.cpp) para pesquisar um arquivo na unidade atual. No entanto, se você estiver projetando um aplicativo que somente os usuários do Windows NT 4.0 usarão, você poderá, em vez disso, usar a função *FindFirstFileEx* (junto com *FindNextFile*) para pesquisar um diretório. A função *FindFirstFileEx* pesquisa em um diretório um arquivo cujo nome e atributos são iguais aos que você especificar na chamada da função. Você usará a função *FindFirstFileEx*, como mostrado no protótipo a seguir:

```
HANDLE FindFirstFileEx(
   LPCTSTR lpNomeArq,          // ponteiro para o nome do arquivo a pesquisar
   FINDEX_INFO_LEVELS fIdNivelInfo,
                       // nível de informações dos dados retornados
   LPVOID lpDadosArqLocaliz, // ponteiro para as informações retornadas
   FINDEX_SEARCH_OPS fOpPesquisa,  // tipo de filtragem a efetuar
   LPVOID lpFiltroPesquisa,  // ponteiro para critério de pesquisa
   DWORD dwSinalizAdic       // sinalizadores adicionais
                             // de controle da pesquisa
);
```

O parâmetro *lpNomeArq* aponta para uma string terminada por *NULL* que especifica um diretório válido ou um nome de arquivo exatamente como com a função *FindFirstFile*, exceto que o parâmetro *lpNomeArq* pode conter caracteres-chave (* e ?). O parâmetro fIdNivelInfo especifica um tipo de enumeração *FINDEX_INFO_LEVELS* que dá o nível de informações dos dados retornados. O parâmetro *lpDadosArqLocaliz* retorna um ponteiro para os dados do arquivo. O nível de informações especificado no parâmetro *fIDNivelInfo* determina o tipo do ponteiro. O parâmetro *fOpPesquisa* especifica um tipo de enumeração *FINDEX_SEARCH_OPS* que informa à função o tipo de filtragem a executar além da comparação dos caracteres-chave. Se o tipo de enumeração que *fOpPesquisa* especifica precisar de informações de pesquisa estruturada, *lpFiltroPesquisa* apontará para o critério de pesquisa. Atualmente, nenhum dos valores de parâmetros *fOpPesquisa* suportados requer informações de pesquisa

estendidas (embora as versões futuras do Windows NT possam requerer informações estendidas). Portanto, esse ponteiro precisa ser *NULL*. O parâmetro *dwSinalizAdic* especifica sinalizadores adicionais para controlar a pesquisa. Você pode usar o sinalizador *FIND_FIRST_EX_CASE_SENSITIVE* para as pesquisas onde o tipo da letra é distingue. A pesquisa normalmente não distingue a caixa das letras. Embora o Windows NT 4.0 não defina outros sinalizadores, as versões futuras do Windows NT poderão definir outros sinalizadores.

Se a função for bem-sucedida, ela retornará um indicativo de pesquisa que seu programa poderá usar em uma chamada subseqüente às funções *FindNextFile* ou *FindClose*. Se a função falhar, ela retornará *INVALID_HANDLE_VALUE*. A função *FindFirstFileEx* lhe permite abrir um indicativo de pesquisa e retornar informações sobre o primeiro arquivo cujo nome é igual ao padrão, e atributos que você especifica na chamada à função. Se o sistema de arquivos subjacente não suportar o tipo de filtragem que o parâmetro *fOpPesquisa* especifica, além de filtragem do diretório, *FindFirstFileEx* falhará, e a função retornará o código de erro *ERROR_NOT_SUPPORTED*. O programa precisará então usar a função *FileExSearchNameMatch* e efetuar sua própria filtragem. Para maiores informações sobre a função *FileExSearchNameMatch*, veja a documentação online do seu compilador.

Quando você estabele o indicativo de pesquisa, seu programa pode usá-lo na função *FindNextFile* para pesquisar outros arquivos que atendam ao mesmo padrão com as funções que efetuem a mesma filtragem. Quando seu programa não precisa mais do indicativo de pesquisa, ele deve fechá-lo usando a função *FindClose*.

Nota: O SDK do Windows NT 4.0 explica os tipos enumerados para FindFirstFileEx em detalhes.

USANDO SEARCHPATH EM VEZ DE FIND PARA LOCALIZAR 1478

Seus programas podem usar as funções *Find* para pesquisar em um diretório uma série de arquivos. Alternativamente, seus programas podem usar a função *SearchFile* para procurar o arquivo que você especificar. No entanto, a função *SearchPath* somente pesquisará o arquivo dentro de um certo conjunto de caminhos, como detalha a Tabela 1478. Você usará a função *SearchPath* dentro de seus programas, como segue:

```
DWORD SearchPath(
   LPCTSTR lpCaminho,  // endereço do caminho de pesquisa
   LPCTSTR lpNomearq,  // endereço do nome de arquivo
   LPCTSTR lpExtensao,  // endereço da extensão
   DWORD nBufferTamanho,// tamanho do buffer em caracteres
   LPTSTR lpBuffer,     // endereço do buffer para o nome
                        // de arquivo encontrado
   LPTSTR *lpParteArq   // endereço do ponteiro para o
                        // componente de arquivo
);
```

A função *SearchPath* aceita os parâmetros detalhados na Tabela 1478.

Tabela 1478 Parâmetros para a função ***SearchPath***.

Parâmetro	Descrição
lpCaminho	Aponta para uma string terminada por *NULL* que especifica o caminho que a função pesquisa para o arquivo. Se esse parâmetro for *NULL*, a função pesquisará uma ocorrência do arquivo nos seguintes diretórios na seguinte seqüência: 1. O diretório a partir do qual o aplicativo foi carregado. 2. O diretório atual. 3. Sob o Windows 95, o diretório do sistema Windows. Use a função *GetSystemDirectory* para obter o caminho desse diretório. Sob o Windows NT, o diretório do sistema Windows de 32 bits. Use a função *GetSystemDirectory* para obter o caminho desse diretório. O nome do diretório do sistema Windows de 32 bits é tipicamente *SYSTEM32*

Tabela 1478 Parâmetros para a função *SearchPath*. (*Continuação*)

Parâmetro	Descrição
	4. Sob o Windows NT, o diretório do sistema do Windows de 16 bits. Não há função Win32 que obtenha o caminho desse diretório, mas a função o pesquisará mesmo assim. O nome desse diretório é tipicamente *SYSTEM*. 5. O diretório do Windows. Use a função GetWindowsDirectory para obter o caminho desse diretório. 6. Os diretórios listados pela variável do ambiente *PATH*.
lpNomeArq	Aponta para uma string terminada por *NULL* que especifica o nome do arquivo a pesquisar.
lpExtensao	Aponta para uma string terminada por *NULL* que especifica uma extensão que a função acrescenta ao nome de arquivo ao pesquisar o arquivo. O primeiro caractere da extensão do nome de arquivo precisa ser um ponto. A função acrescenta a extensão somente se o nome de arquivo que você especificar não terminar com uma extensão. Se o programa não requerer uma extensão de nome de arquivo ou se o nome de arquivo contiver uma extensão, esse parâmetro poderá ser *NULL*.
nBufferTamanho	Especifica o comprimento, em caracteres, do buffer que recebe o caminho o nome de arquivos válidos.
lpBuffer	Aponta para o buffer para o caminho e o nome de arquivo válido do arquivo encontrado.
lpParteArq	Aponta para o endereço (dentro de *lpBuffer*) do último componente do caminho e do nome de arquivo válidos, que é o endereço do caractere que segue imediatamente a barra invertida final (\) no caminho.

Se a função *SearchPath* for bem-sucedida, ela retornará o comprimento, em caracteres, da string que a função copiou para o buffer, não incluindo o caractere *NULL* finalizador. Se o valor de retorno (isto é, o comprimento da string) for maior que *nBufferTamanho*, o valor que a função *SearchPath* retorna será o tamanho do buffer que ela precisará para conter o caminho. Se a função falhar, ela retornará zero. Para obter as informações estendidas de erro, chame *GetLastError*.

O CD-ROM que acompanha este livro inclui o programa *Search_For_Calc.cpp*. Quando você compilar e executar este programa, ele usará a função *SearchPath* para pesquisar a unidade para o arquivo *calc.exe*. Se o programa localizar o arquivo *calc.exe*, ele exibirá o caminho do arquivo em uma caixa de mensagem.

1479 OBTENDO UM CAMINHO TEMPORÁRIO

Nas Dicas 386 e 387 você aprendeu como criar um arquivo temporário dentro de programas C. Como visto, um dos passos que você executará ao criar um arquivo temporário será determinar que o caminho temporário é da variável de ambiente *TEMP* ou *TMP*. No Windows, você pode usar a função *GetTempPath* para recuperar o caminho do diretório que o Windows designa para os arquivos temporários. Você usará a função *GetTempPath* dentro de seus programas, como mostra o protótipo a seguir:

```
DWORD GetTempPath(
   DWORD  nBufferTamanho,    // tamanho do buffer em caracteres
   LPTSTR lpBuffer           // endereço do buffer para o caminho temporário
```

O parâmetro *nBufferTamanho* especifica o tamanho, em caracteres, do buffer de string identificado por *lpBuffer*. O parâmetro *lpBuffer* aponta para um buffer de string que recebe a string terminada por *NULL*, que especifica o caminho do arquivo temporário.

Se a função *GetTempPath* for bem-sucedida, ela retornará o comprimento, em caracteres, da string que o programa copia para o parâmetro *lpBuffer*, não incluindo o caractere *NULL* finalizador. Se o valor de retorno (a string do caminho) for maior que *nBufferTamanho*, o valor de retorno será o tamanho do buffer que a função requererá para conter o caminho. Se a função falhar, ela retornará zero.

A função *GetTempPath* recebe o caminho do arquivo temporário como segue.

1. O caminho que a variável de ambiente *TMP* especifica.
2. O caminho que a variável de ambiente *TEMP* especifica, se o Windows não define *TMP*.
3. O diretório atual, se o Windows não define *TMP* ou *TEMP*.

Criando Arquivos Temporários 1480

Nas Dicas 386 e 387 você aprendeu a criar arquivos temporários a partir de dentro do DOS. Criar um arquivo temporário a partir de dentro de um programa Windows é similarmente fácil. A função *GetTempFileName* cria um nome para um arquivo temporário. O nome de arquivo é a concatenação das strings do caminho e de prefixo que você especificar, uma string hexadecimal formada a partir de um inteiro que você especificar, e a extensão *TMP*. O inteiro que você especifica pode ser diferente de zero; neste caso a função *GetTempFileName* criará o nome de arquivo, mas não criará o arquivo. Se você especificar zero para o inteiro, a função criará um nome de arquivo único e criará o arquivo no diretório especificado. Você usará a função *GetTempFileName* dentro de seus programas, como segue:

```
UINT GetTempFileName(
   LPCTSTR lpNomeCaminho,   // endereço do nome de diretório
                            // para o arquivo temporário
   LPCTSTR lpStringPrefixo,// endereço do prefixo de nome de arquivo
   UINT uUnico,             // número usado para criar nome
                            // de arquivo temporário
   LPTSTR lpNomeArqTemp     // endereço do buffer que recebe
);                          // o novo nome de arquivo
```

O parâmetro *lpNomeCaminho* aponta para uma string terminada por *NULL* que especifica o caminho do diretório para o nome de arquivo. A string terminada por *NULL* precisa consistir de caracteres no conjunto de caracteres ANSI. Os aplicativos tipicamente especificam um ponto ou o resultado da função *GetTempPath* para o parâmetro *lpNomeCaminho*. Se esse parâmetro for *NULL*, a função falhará. O parâmetro *lpStringPrefixo* aponta para uma string de prefixo terminada por *NULL*. A função *GetTempPath* usa os três primeiros caracteres da string de prefixo terminada por *NULL* como o prefixo do nome de arquivo. Essa string precisa consistir dos caracteres no conjunto de caracteres ANSI. O parâmetro *uUnico* especifica um inteiro não-sinalizado que a função converte para uma string hexadecimal para uso na criação do nome de arquivo temporário. Se *uUnico* for diferente de zero, a função acrescentará a string hexadecimal para *lpStringPrefixo* para formar o nome de arquivo temporário. Se *uUnico* for diferente de zero, a função não criará o arquivo especificado, e não testará se o nome de arquivo é único. Se *uUnico* for zero, a função usará valores diferentes até que ela encontre um nome de arquivo único, e, depois, criará o arquivo no diretório *lpNomeCaminho*. O parâmetro *lpNomeArqTemp* aponta para o buffer que recebe o nome de arquivo temporário. Esse buffer é uma string terminada por *NULL* que consiste de caracteres no conjunto de caracteres ANSI. Esse buffer deve ter pelo menos o comprimento, em bytes, que a constante *MAX_PATH* definida pelo sistema especifica (tipicamente 255) para acomodar o caminho.

Se a função *GetTempPath* for bem sucedida, ela retornará o valor numérico único usado no nome de arquivo temporário. Se o parâmetro *uUnico* for diferente de zero, a função retornará esse mesmo número. Se a função falhar, ela retornará zero.

A função *GetTempFileName* cria um nome de arquivo temporário da forma *caminho\preuuuu.TMP*, onde *caminho* representa o caminho que o parâmetro *lpNomeCaminho* especifica, *pre* representa as três primeiras letras da string *lpStringPrefixo*, e *uuuu* representa o valor hexadecimal do parâmetro *uUnico*. Quando o Windows for finalizado, ele não excluirá automaticamente os arquivos temporários cujos nomes a função *GetTempFileName* criou.

Se o parâmetro *uUnico* for zero, *GetTempFileName* tentará formar um número único com base na hora atual do sistema. Se um arquivo com o nome resultante existe, *GetTempFileName* incrementa o número e repete o teste para a existência do nome de arquivo. A função continua seu teste até encontrar um nome de arquivo único. A função *GetTempFileName* então cria um arquivo por aquele nome único e o fecha. Quando *uUnico* for diferente de zero, a função não tentará criar e abrir o arquivo.

O CD-ROM que acompanha este livro inclui o programa *Create_Temp.cpp*, que usa as funções *GetTempPath* e *GetTempFileName* para obter um nome de arquivo temporário. Quando o programa inicia, ele cria um arquivo temporário para uso do programa, e exibe uma caixa de mensagem com o nome do arquivo temporário. Quando o programa termina, ele fecha e apaga o arquivo.

1481 INTRODUZINDO A FUNÇÃO CREATENAMEDPIPE

Como você aprendeu, seus programas podem usar canalizações (que geralmente você usará para comunicação entre dois ou mais computadores) em um modo similar à entrada e saída de caracteres. Quando você quiser usar uma canalização dentro de seus programas, precisará primeiro criá-la. A função *CreateNamedPipe* cria uma ocorrência de uma canalização nomeada em um servidor de canalização e retorna um indicativo para operações de canalização subseqüentes. Um processo de servidor de canalização nomeado usa a função *CreateNamedPipe* ou para criar a primeira ocorrência de uma canalização nomeada que você especifica e estabelecer seus atributos básicos ou para criar uma nova ocorrência de uma canalização nomeada existente. Observe que você somente pode usar canalizações nomeadas para se comunicar em uma rede. Você usará a função *CreateNamedPipe*, como mostra o protótipo a seguir:

```
HANDLE CreateNamedPipe(
   LPCTSTR lpNome,     // ponteiro para o nome da canalização
   DWORD dwModoAbert, // modo de abertura da canalização
   DWORD dwModoCanal, // modos específicos de canalização
   DWORD nMaxOcor,     // números máximos de ocorrências
   DWORD nTamBufferSaida, // tamanho do buffer de saída, em bytes
   DWORD nTamBufferEnt,   // tamanho do buffer de entrada, em bytes
   DWORD nTempoLimPadrao, // tempo-limite, em milissegundos
   LPSECURITY_ATTRIBUTES lpAtribSegur  // ponteiro para
                                       // atributos de segurança
     );
```

A função *CreateNamedPipe* aceita os parâmetros detalhados na Tabela 1481.1.

Tabela 1481.1 *Os parâmetros para a função* ***CreateNamedPipe****.*

Parâmetros	Descrição
lpNome	Aponta para a string terminada por *NULL* que identifica de forma exclusiva a canalização. A string precisa ter a forma \\.*canaliz**nomecanaliz*. A seção *nomecanaliz* do nome pode incluir qualquer caractere diferente de uma barra invertida, incluindo números e caracteres especiais. A string inteira do nome da canalização pode ter até 256 caracteres. Os nomes de canalização não fazem distinção entre os tipos de letras.
dwModoAbert	Especifica o modo de acesso da canalização, o modo de sobreposição, o modo de escrita e o modo de acesso de segurança do indicativo da canalização. O parâmetro *dwModoAbert* precisa especificar um dos sinalizadores de modo de acesso da canalização detalhados na Tabela 1481.2, e precisa especificar o mesmo sinalizador de modo para cada ocorrência da canalização.
dwModoCanaliz	Especifica o tipo e modos de leitura e de espera do indicativo da canalização. O parâmetro *dwModoCanaliz* precisa especificar um ou mais dos sinalizadores do modo de tipo que a Tabela 1481 3 detalha, e precisa especificar o mesmo modo ou modos de tipo para cada ocorrência da canalização. Se você especificar zero, o parâmetro assume por padrão o modo do tipo-byte.
nMaxOcor	Especifica o número máximo de ocorrências que a função pode criar para essa canalização. O parâmetro *nMaxOcor* precisa especificar o mesmo número para todas as ocorrências. Os valores aceitáveis estão no intervalo de 1 até *PIPE_UNLIMITED_INSTANCES*. Se esse parâmetro for *PIPE_UNLIMITED_INSTANCES*, somente a disponibilidade de recursos do sistema limita o número de ocorrências de canalização que *CreateNamedPipe* pode criar.

Tabela 1481.1 Os parâmetros para a função ***CreateNamedPipe***. (*Continuação*)

Parâmetros	Descrição
nTamBufferSaida	Especifica o número de bytes a reservar para o buffer de saída.
nTamBufferEnt	Especifica o número de bytes a reservar para o buffer de entrada.
nTempoLimPadrao	Especifica o valor de tempo-limite padrão, em milissegundos, se a função *WaitNamedPipe* especifica *NMPWAIT_USE_DEFAULT_WAIT*. Cada ocorrência de uma canalização nomeada precisa especificar o mesmo valor.
lpAtribSegur	Ponteiro para uma estrutura *SECURITY_ATTRIBUTES* que especifica um descritor de segurança para a nova canalização nomeada e determina se os processos-filho podem herdar o indicativo retornado. Se *lpAtribSegur* for *NULL*, a canalização nomeada obterá um descritor de segurança padrão, e os processos-filho não poderão herdar o indicativo.

Como você aprendeu na Tabela 1481.1, existem diversos valores constantes predefinidos que o Windows lhe permite usar para o parâmetro *dwModoAbertura*. A Tabela 1481.2 lista os possíveis modos de valores.

Tabela 1481.2 Os valores possíveis para o parâmetro ***dwModoAbert***

Modo	Descrição
FILE_FLAG_WRITE_THROUGH	Habilita o modo de escrita. Esse modo afeta somente as operações de escrita em canalizações do tipo byte e, depois, somente quando os processos cliente e servidor estiverem em computadores diferentes. Se *FILE_FLAG_WRITE_THROUGH* habilita esse modo, as funções que escrevem em uma canalização nomeada não retornam até que o sistema transmita os dados escritos na rede e coloca-os no buffer da canalização no computador remoto. Se a chamada à ***CreateNamedPipe*** não habilitar esse modo, o sistema expandirá a eficiência das operações de rede bufferizando dados até que um número mínimo de bytes se acumulem ou até que um tempo máximo transcorra.
FILE_FLAG_OVERLAPPED	Habilita o modo de sobreposição. Se a função *CreateNamedPipe* habilita esse modo, as funções que efetuam operações de leitura e escrita e as operações de conexão que podem levar um tempo significativo para completar podem retornar imediatamente. Por exemplo, no modo sobreposto, um encadeamento pode tratar simultaneamente as operações de entrada e de saída em múltiplas ocorrências de uma canalização ou efetuar operações simultâneas de leitura e de escrita no mesmo indicativo de canalização. Se a função *CreateNamedPipe* não habilitar o modo de sobreposição, as funções que executam as operações de leitura, gravação e operações de conexão no indicativo de canalização não retornarão até que o sistema operacional finalize a operação. Um programa somente pode usar *ReadFileEx* e *WriteFileEx* com um indicativo de canalização no modo de sobreposição. As funções *ReadFile*, *WriteFile*, *ConnectNamedPipe* e *TransactNamedFile* apenas podem executar sincronamente ou como operações sobrepostas. Esse parâmetro pode incluir qualquer combinação dos sinalizadores de modo de acesso de segurança que esta tabela detalha. Esses sinalizadores de modo de acesso podem ser diferentes para ocorrências diferentes da mesma canalização. Você pode especificá-los sem preocupação com o modo *dwModoAbertura* que você já especificou.

Tabela 1481.2 Os valores possíveis para o parâmetro ***dwModoAbert***. (*Continuação*)

Modo	Descrição
WRITE_DAC	O processo chamador terá acesso de escrita para a lista de controle do acesso discricionário (ACL) da canalização nomeada.
WRITE_OWNER	O processo chamador terá acesso de escrita para o proprietário da canalização
ACCESS_SYSTEM_SECURITY	O processo chamador terá acesso de gravação à ACL do sistema da canalização nomeada.

O parâmetro *dwModoCanaliz* lhe permite especificar como você quer usar a ocorrência atual da canalização. Você pode especificar o tipo, o modo real e o modo de espera dentro do parâmetro *dwModoCanaliz*. A Tabela 1481.3 lista os valores de modo possíveis.

Tabela 1481.3 Valores possíveis para o parâmetro ***dwModoCanaliz***.

Modo	Descrição
PIPE_TYPE_BYTE	Escreve dados na canalização como um fluxo (*stream*) de bytes. Você não pode usar esse modo com *PIPE_READMODE_MESSAGE*.
PIPE_TYPE_MESSAGE	Escreve dados na canalização como um fluxo de mensagens. Você pode usar esse modo com *PIPE_READMODE_MESSAGE* ou com *PIPE_READMODE_BYTE*.
PIPE_READMODE_BYTE	Lê dados da canalização como um bloco de bytes. Você pode usar esse modo com *PIPE_TYPE_MESSAGE* ou *PIPE_TYPE_BYTE*. Você pode especificar diferentes modos de leitura para diferentes ocorrências da mesma canalização. Se você especificar zero, o parâmetro assumirá o modo de leitura de byte.
PIPE_READMODE_MESSAGE	Lê dados da canalização como um bloco de mensagens. Você somente pode usar esse modo se também especificar *PIPE_TYPE_MESSAGE*. Você pode especificar diferentes modos de leitura para diferentes ocorrência da mesma canalização. Se você especificar zero, o parâmetro assumirá por padrão o modo de leitura de byte.
PIPE_WAIT	Habilita o modo de bloco. Quando você especifica o indicativo de canalização nas funções *ReadFile*, *WriteFile* ou *ConnectNamedPipe*, o sistema operacional não completa as operações até que a função leia dados, grave todos os dados ou conecte um cliente, respectivamente. O uso do modo *PIPE_WAIT* pode significar esperar indefinidamente em algumas situações que um processo-cliente efetue uma ação. Você pode especificar diferentes modos de espera para diferentes ocorrências da mesma canalização. Se você especificar zero, o parâmetro assumirá por padrão o modo de bloco.
PIP_NOWAIT	Habilita o modo não de bloco. Neste modo, *ReadFile*, *WriteFile* e *ConnectNamedFile* sempre retornam imediatamente.
PIPE_ACCESS_DUPLEX	A canalização é bidirecional; tanto os processos servidor quanto cliente podem ler e escrever na canalização. Esse modo dá ao servidor o equivalente do acesso *GENERIC_READ* | *GENERIC_WRITE* à canalização. O cliente pode especificar *GENERIC_READ* ou *GENERIC_WRITE*, ou ambos, quando ele se conecta à canalização usando a função *CreateFile*. Você pode especificar modos de acesso diferentes para diferentes ocorrências da mesma canalização.
PIPE_ACCESS_INBOUND	O fluxo de dados na canalização vai do cliente para o servidor somente. Esse modo dá ao servidor o equivalente do acesso *GENERIC_READ* à canalização. O cliente precisará especificar o acesso *GENERIC_WRITE* quando ele se conectar à canalização

Tabela 1481.3 Valores possíveis para o parâmetro **dwModoCanaliz** (*Continuação*)

Modo	Descrição
PIPE_ACCESS_OUTBOUND	O fluxo de dados na canalização vai do servidor para o cliente somente. Esse modo dá ao servidor o equivalente do acesso *GENERIC_WRITE* à canalização. O cliente precisa especificar o acesso *GENERIC_READ* quando ele se conectar à canalização.

Se a função *CreateNamedPipe* for bem-sucedida, ela retornará um indicativo para a ponta do servidor de uma ocorrência de canalização nomeada. Se a função falhar, ela retornará *INVALID_HANDLE_VALUE*. Para obter informações de erro estendidas, chame *GetLastError*. A função retornará *ERROR_INVALID_PARAMETER* se o parâmetro *nMaxOcorrencias* for maior que *PIPE_UNLIMITED_INSTANCES*.

Para usar *CreateNamedPipe* para criar uma ocorrência de uma canalização nomeada, o usuário precisa ter o acesso *FILE_CREATE_PIPE_INSTANCE* ao objeto da canalização nomeado. Se a função *CreateNamedPipe* estiver criando uma nova canalização nomeada, a lista de controle de acesso (ACL) do parâmetro dos atributos de segurança definirá o controle de acesso discricionário para a canalização nomeada.

Todas as ocorrências de uma canalização nomeada precisam especificar o mesmo tipo de canalização (tipo byte ou tipo mensagem), acesso da canalização (dúplex, entrada, ou saída), contador da ocorrência e o valor do tempo-limite. Se as ocorrências usam diferentes valores, a função *CreateNamedPipe* falha e *GetLastError* retorna *ERROR_ACCESS_DENIED*.

Os tamanhos do buffer de entrada e saída são utilizados como orientação. Em outras palavras, o tamanho real do buffer que o Windows reserva para cada ponta da canalização nomeada é o padrão do sistema, o mínimo ou o máximo do sistema, ou o tamanho que você especificar arredondado para cima até o próximo limite de alocação. O servidor de canalização não deverá efetuar uma operação de leitura de bloco até que a canalização cliente tenha iniciado. Caso contrário, ocorrerá uma condição de disputa. Tipicamente, a condição de disputa ocorre quando o código de iniciação (tal como uma rotina de iniciação de processo) precisa travar e examinar os indicativos herdados. Uma *condição de disputa* é um erro em um processo multiencadeado onde o código de um encadeamento depende de um segundo encadeamento para completar alguma ação, mas onde não há sincronização entre os dois encadeamentos. O processo funcionará se o segundo encadeamento "ganhar" a disputa completando sua ação antes do primeiro encadeamento precisar do segundo encadeamento, mas falhará se o primeiro encadeamento "ganhar" a disputa.

O programa sempre exclui uma ocorrência de uma canalização nomeada quando fecha o último encadeamento para aquela ocorrência da canalização nomeada. Na Dica 1483, logo à frente, você usará *CallNamedPipe* e *ConnectNamedPipe* junto com *CreateNamedPipe* para efetuar saída em um servidor de rede.

CONECTANDO UMA CANALIZAÇÃO NOMEADA 1482

Seus programas podem usar a função *CreateNamedPipe* para criar ou se conectar a uma canalização nomeada. No entanto, muitas vezes, seu programa rodará na máquina cliente, o que o forçará a se conectar a uma canalização nomeada para se comunicar com o servidor, em vez de criar uma canalização nomeada (o que precisará ocorrer no servidor da canalização). Além disso, antes de seu programa poder se conectar à canalização nomeada do servidor, a canalização nomeada precisará chamar a função *ConnectNamedPipe*, que instruirá o processo servidor da canalização nomeada para esperar um processo cliente para se conectar a uma ocorrência de uma canalização nomeada. Um processo cliente chamará a função *CreateFile* ou *CallNamedPipe* para se conectar à ocorrência. Você aprenderá sobre a função *CallNamedPipe* na dica a seguir. Você usará a função *ConnectNamedPipe* dentro de seus programas, como mostrado aqu.:

```
BOOL ConnectNamedPipe(
   HANDLE nCanalizNomeada, // indicativo para canalização
                           // nomeada a conectar
   LPOVERLAPPED lpSobreposta // ponteiro para estrutura sobreposta
```

O parâmetro *hCanalizNomeada* identifica a ponta do servidor de uma ocorrência de canalização nomeada. A função *CreateNamedPipe* retorna esse indicativo. O parâmetro *lpSobreposta* aponta para uma estrutura *OVERLAPPED* (que, como você aprendeu na Dica 1451, permite que seus programas efetuem E/S assíncrona). Se a

função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, a função retornará zero. Para obter informações adicionais sobre o erro, chame *GetLastError*.

Um processo servidor de canalização nomeada pode usar *ConnectNamedPipe* com uma ocorrência de canalização recém-criada ou com uma ocorrência que foi anteriormente conectada a outro processo cliente. Nesse caso, o processo servidor precisará primeiro chamar a função *DisconnectNamedPipe* para desconectar o indicativo do cliente anterior antes de poder reconectar o indicativo a um novo cliente. Caso contrário, *ConnectNamedPipe* retorna *False*, e *GetLastError* retorna *ERROR_NO_DATA*, se o cliente anterior fechou seu indicativo, ou *ERROR_PIPE_CONNECTED*, se ele não fechou seu indicativo.

O comportamento da função *ConnectNamedPipe* depende de duas condições: se você definiu o modo de espera do indicativo da canalização como de bloco ou não de bloco e se você definiu a função para executar sincronamente ou no modo sobreposto. Um servidor inicialmente especifica o modo de espera de indicativo da canalização na função *CreateNamedPipe*, e você usa a função *SetNamedPipeHandleState* para alterá-lo.

Se a função abrir *nCanalizNomeada* com *FILE_FLAG_OVERLAPPED*, o parâmetro *lpSobreposta* não poderá ser *NULL*. Ele precisará apontar para uma estrutura *OVERLAPPED* válida. Se a função abrir *hCanalizNomeada* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* for *NULL*, a função *ConnectNamedPipe* poderá incorretamente informar que a operação de conexão está completa. Por outro lado, se a função criar *hCanalizNomeada* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* não for *NULL*, a estrutura *OVERLAPPED* para a qual o parâmetro *lpSobreposta* aponta precisará conter um indicativo para um objeto de evento de reiniciação manual (que o servidor pode criar com a função *CreateEvent*).

Se a função não abrir *hCanalizNomeada* com *FILE_FLAG_OVERLAPPED* e *lpSobreposta* for *NULL*, a função não retornará até que o programa conecte um cliente ou até que ocorra um erro. As operações síncronas resultam na função retornando *True* se um cliente se conectar após o programa chamar a função. Se um cliente se conectar antes de o programa chamar a função, a função retornará *False*, e *GetLastError* retornará *ERROR_PIPE_CONNECTED*. Um resultado *False* poderá acontecer se um cliente se conectar no intervalo entre a chamada à *CreateNamedPipe* e a chamada à *ConnectNamedPipe*. Nesta situação, há uma boa conexão entre o servidor e o cliente, embora a função retorne *False*.

Se a função não abrir *hCanalizNomeada* com *FILE_FLAG_OVERLAPPED*, e *lpSobreposta* for *NULL*, a operação executará assincronamente. A função retorna imediatamente com um valor *False*. Se um processo cliente se conectar antes de o programa chamar a função, *GetLastError* retornará *ERROR_PIPE_CONNECTED*. Caso contrário, *GetLastError* retornará *ERROR_IO_PENDING*, o que indica que a operação está executando no segundo plano. Quando isso acontece, o programa define o objeto evento na estrutura *OVERLAPPED* com o estado não-sinalizado antes que *ConnectNamedPipe* retorne, e o programa a define para o estado sinalizado quando um cliente se conecta a essa ocorrência da canalização.

O processo servidor pode usar qualquer uma das funções de espera ou a função *SleepEx* para determinar quando o estado do objeto evento está sinalizado. Então, o servidor pode usar a função *GetOverlappedResult* para determinar os resultados da operação *ConnectNamedPipe*. Se o indicativo de canalização que você especificar estiver no modo não de bloco, *ConnectNamedPipe* sempre retornará imediatamente. No modo não de bloco, *ConnectNamedPipe* retornará *True* na primeira vez em que o programa a chamar para uma ocorrência de canalização que o processo já desconectou de um cliente anterior. Isso indica que a canalização está agora disponível para o sistema conectá-la a um novo processo cliente. Em todas as outras situações, quando o indicativo da canalização estiver no modo não de bloco, *ConnectNamedPipe* retornará *False*. Nessas situações, *GetLastError* retornará *ERROR_PIPE_LISTENING* se nenhum cliente estiver conectado, *ERROR_PIPE_CONNECTED* se um cliente estiver conectado e *ERROR_NO_DATA* se um cliente anterior fechou seu indicativo de canalização mas o servidor não se desconectou. Note, quando uma canalização está no modo não de bloco, uma boa conexão entre cliente e servidor existe somente após o programa receber um error *ERROR_PIPE_CONNECTED*.

Nota: ConnectNamedPipe suporta o modo não de bloco para compatibilidade com o LAN Manager 2.0 da Microsoft. Você não deve usá-lo para alcançar E/S assíncrona com as canalizações nomeadas.

1483 CHAMANDO UMA CANALIZAÇÃO NOMEADA

Seus programas podem criar uma canalização nomeada na ponta do servidor de uma conexão de rede. A ponta do cliente precisa então se conectar à canalização nomeada para enviar informações para a canalização. Dentro de seus programas, você pode usar *CreateFile* ou *CallNamedPipe* para se conectar a uma canalização nomeada. A função *CallNamedPipe* se conecta a uma canalização de tipo de mensagem (e espera se uma ocorrência da ca-

nalização não estiver disponível), escreve e lê na canalização e depois a fecha. Você usará a função *CallNamedPipe*, como segue:

```
BOOL CallNamedPipe(
   LPCTSTR lpNomeCanalizNomeada, // ponteiro para o nome
   LPVOID lpBufferEnt,           // ponteiro para o buffer de escrita
   DWORD nTamBufferEnt,    // tamanho, em bytes, do buffer de escrita
   LPVOID lpBufferSaida,   // ponteiro para o buffer de leitura
   DWORD nTamBufferSaida,  // tamanho, em bytes, do buffer de leitura
   LPDWORD lpBytesLidos,   // ponteiro para o número de bytes lidos
   DWORD nTempoLim     // tempo-limite, em milissegundos
);
```

A função *CallNamedPipe* aceita os parâmetros que a Tabela 1483.1 detalha.

Tabela 1483.1 Parâmetros para a função ***CallNamedPipe***.

Parâmetro	Descrição
lpNomeCanalizNomeada	Ponteiro para uma string terminada por *NULL* que especifica o nome da canalização.
lpBufferEnt	Ponteiro para o buffer que contém os dados que *CallNamedPipe* escreve na canalização.
nTamBufferEnt	Especifica o tamanho em bytes do buffer de escrita.
lpBufferSaida	Ponteiro para o buffer que recebe os dados que *CallNamedPipe* lê da canalização.
nTamBufferSaida	Especifica o tamanho em bytes do buffer de leitura.
nBytesLidos	Ponteiro para uma variável de 32 bits que recebe o número de bytes que *CallNamedPipe* lê da canalização.
nTempoLim	Especifica o número de milissegundos a esperar até que a canalização nomeada fique disponível. Além dos valores numéricos, seus programas podem especificar os valores que a Tabela 1483.2 lista.

Quando você chamar uma canalização nomeada em uma rede, é importante que coloque um limite no tempo que seus programas esperarão por uma resposta da canalização nomeada. O tempo-limite evita que os computadores clientes aguardem indefinidamente uma conexão de canalização nomeada que pode estar ocupada ou até mesmo ser não-existente. Além de especificar um número fixo de milissegundos para seus programas aguardarem por uma resposta da canalização nomeada, você pode usar os valores que a Tabela 1483.2 detalha para fornecer controle sobre o tempo que seus processos clientes aguardam uma conexão de canalização nomeada.

Tabela 1483.2 Constantes possíveis de duração da espera para a função ***CallNamedPipe***

Valor	Significado
NMPWAIT_NOWAIT	Não espera pela canalização nomeada. Se a canalização nomeada não estiver disponível, a função retornará um erro
NMPWAIT_WAIT_FOREVER	Aguarda indefinidamente.
NMPWAIT_USE_DEFAULT_WAIT	Usa o tempo-limite padrão especificado em uma chamada à função *CreateNamedPipe*.

Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, ela retornará zero. Para obter as informações estendidas de erro, chame *GetLastError*.

Chamar *CallNamedPipe* é equivalente a chamar as funções *CreateFile* (ou *WaitNamedPipe*, se *CreateFile* não puder abrir a canalização imediatamente), *TransactNamedPipe* e *CloseHandle*. O programa chama *CreateFile* com um sinalizador de acesso de *GENERIC_READ | GENERIC_WRITE*, um sinalizador de indicativo de herança *False* e um modo de compartilhamento de zero (o que indica nenhum compartilhamento da ocorrência dessa canalização). Se a mensagem que o processo servidor escreve na canalização for maior que *nTamBufferSaida*, *CallNamedPipe* retornará *False*, e *GetLastError* retornará *ERROR_MORE_DATA*. O processo servidor descarta o restante da mensagem, pois *CallNamedPipe* fecha o indicativo para a canalização antes de retornar.

O CD-ROM que acompanha este livro inclui os programas *Server.cpp* e *Client.cpp*. Quando você compilar e executar o programa *Server.cpp*, ele criará uma canalização nomeada na máquina local. Se você, em seguida, compilar e executar o programa *Client.cpp*, ele chamará a canalização nomeada. *Client.cpp* exibirá a mensagem, e, depois, escreverá uma string de dados na canalização nomeada. Quando o servidor receber a mensagem, ele exibirá as informações dentro de seu próprio processo. Observe que este exemplo roda em uma única máquina atuando como cliente e como servidor.

Nota: A função CallNamedPipe falhará se a canalização que a função tentar chamar for uma canalização do tipo byte.

1484 DESCONECTANDO UMA CANALIZAÇÃO NOMEADA

Como você aprendeu, seus programas podem usar canalizações nomeadas para comunicação entre dois processos rodando em diferentes máquinas (e até mesmo na mesma máquina). No entanto, após você completar o compartilhamento de informações entre os dois processos, é importante fechar a canalização nomeada para que ela não torne a rede ou o computador mais lentos. A função *DisconnectNamedPipe* desconecta a ponta do servidor de uma ocorrência de canalização nomeada de um processo cliente. Você usará a função *DisconnectNamedPipe*, como segue:

```
BOOL DisconnectNamedPipe(HANDLE hCanalizNomeada);
```

O parâmetro *nCanalizNomeada* identifica uma ocorrência de uma canalização nomeada. A função *CreateNamedPipe* precisa criar esse indicativo. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, ela retornará zero. Para obter informações adicionais sobre o erro, chame *GetLastError*.

Se a ponta do cliente de uma canalização nomeada estiver aberta, a função *DisconnectNamedPipe* forçará essa ponta a fechar. O cliente recebe um erro na próxima vez em que tentar acessar a canalização. Um cliente que *DisconnectNamedPipe* forçar a partir de uma canalização ainda precisará usar a função *CloseHandle* para fechar sua ponta da canalização.

Quando o processo servidor desconecta uma ocorrência de canalização, ele descarta quaisquer dados não-lidos na canalização. Antes de se desconectar, para garantir que não há perda de dados, o servidor pode chamar a função *FlushFileBuffers*, que não retorna até que o processo cliente tenha lido todos os dados. O processo servidor precisa chamar *DisconnectNamedPipe* para desconectar um indicativo de canalização de seu cliente anterior antes de poder conectar o indicativo para outro cliente usando a função *ConnectNamedPipe*.

1485 COMPREENDENDO MELHOR O PROCESSAMENTO ASSÍNCRONO

Seus programas podem armazenar e recuperar informações dos arquivos na maioria dos dispositivos usando saída síncrona e assíncrona. Como você também aprendeu, a E/S assíncrona de arquivo permite que seus programas efetuem operações de arquivo não-sincronizadas. No entanto, é importante compreender melhor por que você poderá querer efetuar E/S assíncrona.

Em comparação à maioria das operações que seu computador executa, a E/S em dispositivo é uma das mais lentas. A CPU executa operações matemáticas e escreve na tela muito mais rapidamente que lê ou grava dados em um arquivo ou na rede. A E/S assíncrona lhe permite usar múltiplos encadeamentos para dizer ao sistema operacional para ler ou escrever em um dispositivo enquanto o restante do código dentro de seu aplicativo continua a executar.

Para compreender melhor como a E/S assíncrona pode melhorar o desempenho do programa, suponha que você esteja desenvolvendo um aplicativo de banco de dados simples. Quando o usuário abrir o banco de dados, o aplicativo lerá o conteúdo do banco de dados na memória, bem como em um arquivo de índice. Após o usuário selecionar o banco de dados a carregar, o aplicativo precisará então fazer uma pausa e carregar todos os dados na memória (provavelmente exibindo uma ampulheta).

No entanto, se você usar E/S assíncrona, um programa pode iniciar a operação de E/S no disco, que a controladora de disco executará, deixando que a CPU execute outras tarefas não-relacionadas ao mesmo tempo. A E/S assíncrona em arquivo lhe permite executar múltiplas atividades de E/S ao mesmo tempo — ou inicie uma atividade de E/S não-crítica e a deixe completar em seu próprio tempo sem reduzir a velocidade de execução do seu programa. À medida que seus programas se tornarem mais complexos, e você for lendo e gravando mais dados do disco rígido ou de outras mídias de armazenamento, os benefícios da E/S assíncrona aumentarão para você.

USANDO ENTRADA E SAÍDA ASSÍNCRONA 1486

Para acessar um dispositivo assincronamente, você precisa primeiro chamar *CreateFile* para abrir o dispositivo e especificar o sinalizador *FILE_FLAG_OVERLAPPED* no parâmetro *dwFlagsAndAttrs*. O sinalizador *FILE_ FLAG_ OVERLAPPED* notifica o sistema de que você pretende usar o dispositivo para E/S assíncrona.

O sistema operacional Win32 lhe permite usar quatro técnicas diferentes para efetuar E/S, assíncrona. As quatro técnicas compartilham uma teoria de operação comum. Quando você efetua E/S assíncrona, precisa primeiro emitir uma solicitação de E/S ao sistema operacional. O sistema operacional enfileirará todas as solicitações de E/S e as tratará internamente. Enquanto o sistema operacional estiver tratando as solicitações de E/S ele permite que seu encadeamento retorne e continue processando. Em algum ponto daí para a frente, o sistema operacional completará a tarefa de E/S e notificará seu aplicativo de que ele enviou e recebeu os dados, ou de que um erro ocorreu.

Como observado anteriormente, existem quatro técnicas diferentes. A Tabela 1486 as lista em ordem de complexidade, da mais fácil de compreender e de implementar (sinalização de indicativo de dispositivo) até a mais difícil de compreender e de implementar (portas de finalização de E/S).

Tabela 1486 As quatro técnicas do processamento assíncrona da E/S.

Técnica	Descrição
Sinalizar um objeto dispositivo	Usa uma função *Wait* e um indicativo de dispositivo para efetuar E/S assíncrona. Não é útil se você vai efetuar múltiplas solicitações de E/S simultâneas em um único dispositivo. Permite que um encadeamento emita uma solicitação de E/S, e outro encadeamento a processe.
Sinalizar um objeto evento	Usa uma função *WaitforMultipleObjects* e um ou mais objetos evento para efetuar E/S assíncrona. Permite que um encadeamento emita uma solicitação de E/S e outro encadeamento a processe.
E/S alertável	Usa uma mensagem especial para processar as notificações do sistema operacional que um processamento de operação de E/S completou. Oferece mais flexibilidade do que o uso de um objeto de núcleo do evento porque você pode usar funções de callback e especificar processamento de mensagem para responder às informações do sistema operacional sobre a ação de E/S. O encadeamento que emitiu a solicitação de E/S também precisa processar a resposta. Você somente pode usar E/S alertável em sistemas Windows NT.
Portas de finalização de E/S	Usa um *modelo de encadeamento concorrente* (explicado à frente na Dica 1495) para responder a um grande número de solicitações de E/S simultâneas. Permite que um encadeamento emita uma solicitação de E/S e outro a processe. Uma técnica altamenta escalável, as portas de finalização são mais freqüentemente usadas por desenvolvedores profissionais. Você somente pode usar portas de finalização de E/S em sistemas Windows NT.

1487 COMPREENDENDO A ESTRUTURA OVERLAPPED

Como indicou a Tabela 1486, a técnica mais simples para efetuar E/S assíncrona de dispositivo é usar sinalização de indicativo de dispositivo, que você usará na dica a seguir. Para emitir a solicitação de E/S você usa as funções *ReadFile* e *WriteFile* que as Dicas 1456 e 1457 apresentaram. No entanto, para efetuar E/S assíncrona de dispositivo, seus programas precisam passar o endereço de uma estrutura *OVERLAPPED* inicializada como o parâmetro *lpSobreposta*. A API Win32 define a estrutura *OVERLAPPED*, como mostrado aqui:

```
typedef struct _OVERLAPPED {
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvento;
} OVERLAPPED;
```

A Tabela 1487 lista os membros da estrutura *OVERLAPPED*.

Tabela 1487 Os membros da estrutura ***OVERLAPPED***

Membro	Descrição
Internal	Especifica um status dependente do sistema. Este membro é válido quando a função *GetOverlapped* retorna sem definir as informações estendidas de erro para *ERROR_IO_PENDING*. Reservado para uso do sistema operacional.
InternalHigh	Especifica o comprimento dos dados transferidos. Esse membro é válido quando a função *GetOverlappedResult* retorna *True*. Reservado para uso do sistema operacional.
Offset	Especifica a posição do arquivo onde iniciar a transferência. A posição do arquivo é um deslocamento de byte a partir do início do arquivo. O processo chamador define esse membro antes de chamar as funções *ReadFile* e *WriteFile*. O processo chamador ignora o membro ao ler ou escrever para as canalizações nomeadas e dispositivos de comunicação.
OffsetHigh	Especifica a palavra mais significativa do deslocamento de byte onde iniciar a transferência. O processo chamador ignora esse membro ao ler e escrever em canalizações nomeadas e dispositivos de comunicação.
hEvent	Identifica um evento definido para o estado sinalizado quando a transferência tiver sido completada. O processo chamador define esse membro antes de chamar a função *ReadFile*, *WriteFile*, *ConnectNamedPipe* ou *TransactNamedPipe*.

Você pode usar a macro *HasOverlappedCompleted* para determinar se uma operação assíncrona de E/S foi completada. Use a função *Cancello* para cancelar uma operação assíncrona de E/S.

1488 E/S ASSÍNCRONA COM UM OBJETO DE NÚCLEO DO DISPOSITIVO

Como você aprendeu, seus programas podem efetuar E/S assíncrona usando quatro técnicas diferentes, a mais simples das quais é o uso de um objeto de dispositivo de núcleo. Quando você efetua E/S assíncrona com um objeto de dispositivo de núcleo, simplesmente instrui o encadeamento a esperar até que a E/S tenha sido completada. Por exemplo, suponha que você leia a partir de um arquivo com a função *ReadFile*, efetue algum processamento intermediário, mas o programa não possa continuar além de um certo ponto até que a operação de leitura termine. Nesse caso, você pode construir código similar ao mostrado no fragmento a seguir:

```
ReadFile(hArq, bBuffer, sizeof(bBuffer), &dwNumBytesLidos, &Sobreposta);
// processamento aqui
WaitForSingleObject(hArq, INFINITE); // aguarda até que todos os dados
estejam no buffer
```

A função *WaitForSingleObject*, quando você a chama com o indicativo para um dispositivo assíncrono de E/S, esperará até que o sistema operacional tenha completado o processamento apropriado para o dispositivo antes de liberar e permitir que o encadeamento continue. O CD-ROM que acompanha este livro inclui o programa *Simple_ASRead.cpp*, que usa a técnica do objeto de núcleo do dispositivo mostrada dentro desta dica para efetuar E/S assíncrona simples.

COMPREENDENDO AS QUOTAS DE TAMANHO DO WORKING SET 1489

Quando seus programas efetuam E/S assíncrona, o sistema operacional mantém uma lista de solicitações de E/S que estão pendentes. O tamanho da lista é fixado durante a iniciação do sistema operacional. Ocasionalmente, uma solicitação de E/S assíncrona poderá falhar porque a lista de solicitações de E/S pendentes já está cheia. Se a lista estiver cheia quando você emitir outra solicitação, *ReadFile* e *WriteFile* retornarão *False*, e *GetLastError* retornará *ERROR_INVALID_USER_BUFFER* ou *ERROR_NOT_ENOUGH_MEMORY*. Além disso, quando você fizer uma solicitação de E/S, o sistema precisará "bloquear por páginas" o buffer de dados do seu programa. O buffer de dados é parte do *working set* (conjunto funcional) do processo, e cada processo tem um *working set* máximo. O *working set* de um processo é o conjunto de páginas de memória atualmente visíveis para o processo na RAM física. Se você não tiver espaço suficiente no *working set* do seu processo, fazer uma solicitação de E/S falhará, e *GetLastError* retornará *ERROR_NOT_ENOUGH_QUOTA*. Você pode aumentar o tamanho do working set chamando *SetProcessWorkingSetSize*, explicada em detalhes na dica a seguir.

DEFININDO QUOTAS MAIORES E MENORES 1490

Como visto na dica anterior, seus programas podem aumentar os tamanhos do *working set* (o conjunto de páginas de memória atualmente visíveis para o processo na memória física), caso os programas requeiram espaço adicional dentro de seus *working set*. Essas página residem e estão disponíveis para um aplicativo usar sem disparar uma falha de página. O tamanho do *working set* de um processo é especificado em bytes. Os tamanhos máximo e mínimo afetam o comportamento da paginação da memória virtual de um processo. A função *SetProcessWorkingSetSize* definirá os tamanhos máximo e mínimo para um processo que você especificar. Você usará a função *SetProcessWorkingSetSize* dentro de seus programas, como segue:

```
BOOL SetProcessWorkingSetSize(
   HANDLE hProcesso,     // indicativo de abertura para o
                         // processo de interesse
   DWORD dwTamMinimo,    // especifica o tamanho mínimo
                         // para o working set
   DWORD dwTamMaximo,    // especifica o tamanho máximo
                         // para o working set
);
```

O parâmetro *hProcesso* é um indicativo de abertura para o processo cujos tamanhos de *working set* você definirá. Sob o Windows NT, o indicativo precisa ter o direito de acesso *PROCESS_SET_QUOTA*. O parâmetro *dwTamMinimo* especifica o tamanho mínimo para o working set do processo. O gerenciador de memória virtual tenta manter pelo menos esse tanto de memória residente no processo sempre que ele está ativo. O parâmetro *dwTamMaximo* especifica o tamanho máximo do working set para o processo. O gerenciador de memória virtual tenta manter não mais do que esse tanto de memória residente no processo sempre que o processo está ativo e a memória escassa.

Se tanto *dwTamMinimo* quanto *dwTamMaximo* tiverem o valor *0xFFFFFFFF*, a função temporariamente reduzirá o working set do processo que você especificar para zero. Isso basicamente transfere o processo para fora da memória física. Se a função for bem-sucedida, ela retornará um valor diferente de zero. Chame *GetLastError* para obter maiores informações sobre o erro.

Você pode esvaziar o working set do processo especificado utilizando o valor *0xFFFFFFFF* tanto para o tamanho máximo quanto para o mínimo do working set. Se os valores de *dwTamMinimo* e *dwTamMaximo* forem maiores que os tamanhos do working set atuais, o processo especificado precisará ter o privilégio *SE_INC_BASE_PRIORITY_NAME*. O sistema operacional aloca tamanhos de *working set* na base do primeiro que entra, primeiro que sai. Por exemplo, se um aplicativo definir com sucesso 40Mb como o tamanho mínimo

do working set em um sistema de 64Mb de memória e um segundo aplicativo solicitar um tamanho de working set de 40Mb, o sistema operacional negará a solicitação do segundo aplicativo.

Usar a função *SetProcessWorkingSetSize* para definir os tamanhos máximo e mínimo do working set de um aplicativo não garante que o sistema operacional reservará a memória solicitada ou que a memória permanecerá residente o tempo todo. Quando o aplicativo estiver ocioso, ou uma situação de escassez causar uma demanda pela memória, o sistema operacional poderá reduzir o working set do aplicativo. Um aplicativo pode usar a função *VirtualLock* para bloquear intervalos do espaço do endereço virtual do aplicativo na memória; no entanto, isso pode potencialmente degradar o desempenho do sistema.

Quando você aumenta o tamanho do working set de um aplicativo, está tirando memória física do restante do sistema. Isso pode degradar o desempenho dos outros aplicativos e do sistema como um todo. Tirar memória física também pode levar a falhas nas operações que requerem muita memória física; por exemplo, criar processos, encadeamentos e uma reserva para o núcleo. Portanto, você precisa usar a função *SetProcessWorkingSetSize* com cuidado. Você sempre precisa considerar o desempenho do sistema todo quando projetar um aplicativo.

1491 Compreendendo a Função GetLastError

Como você aprendeu nas últimas 240 dicas, pode frequentemente obter maiores informações sobre um erro no programa. Geralmente, seus programas devem chamar a função *GetLastError* para obter maiores informações. A função *GetLastError* retorna o valor do código do último erro do encadeamento. O sistema operacional mantém o código do último erro para cada encadeamento. Os encadeamentos não sobrescrevem o código do último erro uns dos outros. Você usará a função *GetLastError* dentro de seus programas, como mostrado aqui:

```
DWORD GetLastError(void);
```

A função retorna o valor do código do último erro do encadeamento chamador. As funções chamam a função *GetLastError* para definir o valor do código do último erro. Você deve chamar a função *GetLastError* imediatamente quando o valor de retorno de uma função indicar que essa chamada retornará dados úteis (tais como maiores informações sobre o erro), pois algumas funções chamarão *SetLastError(0)* quando forem bem-sucedidas, removendo o código de erro que a função que falhou mais recentemente definiu.

A maioria das funções na API Win32 que define o valor do código do último erro do encadeamento faz isso em caso de falha; poucas funções definem o valor se forem bem-sucedidas. Um código de erro de retorno, tal como *False*, *NULL*, *0xFFFFFFFF*, ou -1, tipicamente indicam falha da função. Os códigos de erro são valores de 32 bits (bit 31 é o mais significativo) Se você estiver definindo um código de erro para seu aplicativo, defina o bit 29 como 1. Definir o bit 29 como 1 indica que um aplicativo definiu o código de erro e garante que seu código de erro não entra em conflito com quaisquer códigos de erro que o sistema operacional define. Você pode usar a função *FormatMessage* para formatar a saída a partir do resultado de uma chamada à *GetLastError*. A dica a seguir explicará a função *FormatMessage*.

O CD-ROM que acompanha este livro inclui o programa *GenerateError.cpp*. Quando você compilar e executar este programa, ele efetuará uma série de tarefas que gera erros e depois exibe o texto dos erros em uma caixa de mensagem.

1492 Formatando as Mensagens de Erro com FormatMessage

Vimos na dica anterior que seus programas podem usar a função *GetLastError* para obter uma representação numérica do último erro de um encadeamento. No entanto, você mais comumente irá querer ver a representação em string do último erro do encadeamento.

A função *FormatMessage* formata uma string de mensagem. Ela requer uma definição de mensagem como entrada. A definição de mensagem pode vir de um buffer passado para a função; ela pode vir de um recurso de tabela de mensagem em um módulo já carregado. Além disso, o processo chamador pode pedir que a função pesquise a definição de mensagem nos recursos da tabela de mensagem do sistema. A função localiza a definição de mensagem em um recurso de tabela de mensagem com base no identificador de uma mensagem e um identificador de linguagem. A função copia o texto da mensagem formatada em um buffer de saída, processando quaisquer seqüências de inserção incorporadas, se solicitada. Você usará a função *FormatMessage* dentro de seus programas, como segue:

```
DWORD FormatMessage(
    DWORD dwSinaliz,    // origem e opções de processamento
    LPCVOID lpOrigem,   // ponteiro para mensagem de origem
    DWORD dwIdMensa,    // identificador da mensagem solicitada
    DWORD dwIdIdioma,   // identificador do idioma para a mensagem solicitada
    LPTSTR lpBuffer,    // ponteiro para buffer da mensagem
    DWORD nTamanho,     // tamanho máximo do buffer da mensagem
    va_list *Argumentos // endereço da matriz de inserções da mensagem
);
```

A Tabela 1492 lista os parâmetros para a função *FormatMessage*.

Tabela 1492.1 Os parâmetros para a função ***FormatMessage***.

Parâmetro	Descrição
dwSinaliz	Contém um conjunto de bits sinalizadores que especifica aspectos do processo de formatação e como interpretar o parâmetro *lpOrigem*. O byte menos significativo de *dwSinaliz* especifica como a função trata as quebras de linha no buffer de saída. O byte menos significativo também pode especificar a largura máxima de uma linha de saída formatada. Você pode especificar uma combinação dos sinalizadores que a Tabela 1491.2 já detalhou.
lpOrigem	Especifica a posição da definição da mensagem. O tipo desse parâmetro depende das definições no parâmetro *dwSinaliz*. Se você definir *dwSinaliz* como *FORMAT_MESSAGE_FROM_HMODULE*, *lpOrigem* será um *hModulo* do módulo que contém a tabela de mensagem a pesquisar. Por outro lado, se você definir *dwSinaliz* como *FORMAT_MESSAGE_FROM_STRING*, *lpOrigem* será um *LPTSTR* que aponta para texto de mensagem não-formatado. Se você não definir um desses sinalizadores em *dwSinaliz*, então a função ignorará *lpOrigem*.
dwIdMensa	Especifica o identificador de mensagem de 32 bits para a mensagem requerida. A função ignorará esse parâmetro se *dwSinaliz* incluir *FORMAT_MESSAGE_FROM_STRING*.
dwIdIdioma	Especifica o identificador de 32 bits do idioma para a mensagem solicitada. A função ignorará esse parâmetro se *dwSinaliz* incluir *FORMAT_MESSAGE_FROM_STRING*. Se você passar um *LANGID* específico nesse parâmetro, *FormatMessage* retornará uma mensagem para esse *LANGID* somente. Se a função não puder localizar uma mensagem para esse *LANGID*, ela retornará *ERROR_RESOURCE_LANG_NOT_FOUND*.
lpBuffer	Aponta para um buffer para a mensagem formatada (e terminada por *NULL*). Se *dwSinaliz* incluir *FORMAT_MESSAGE_ALLOCATE_BUFFER*, a função alocará um buffer usando a função *LocalAlloc*, e colocará o endereço do buffer no endereço especificado em *lpBuffer*.
nTamanho	Se você não definir o sinalizador *FORMAT_MESSAGE_ALLOCATE_BUFFER*, esse parâmetro especificará o número máximo de bytes (versão ANSI) ou caracteres (versão Unicode) que o programa pode armazenar no buffer de saída. Se você definir *FORMAT_MESSAGE_ALLOCATE_BUFFER*, esse parâmetro especificará o número mínimo de bytes ou caracteres a alocar para um buffer de saída.
Argumentos	Aponta para uma matriz de valores de 32 bits que são usados como valores de inserção na mensagem formatada. %1 na string de formato indica o primeiro valor na matriz *Argumentos*; %2 indica o segundo argumento; e assim por diante.

A interpretação de cada valor de 32 bits depende das informações de formatação contidas no parâmetro *dwSinaliz* e da posição da definição de mensagem real. O padrão é tratar cada valor como um ponteiro para uma string terminada por *NULL*. Por padrão, o parâmetro *Argumentos* é do tipo *va_list**, que é um tipo de dado específico da implementação e da linguagem para descrever um número variável de argumentos. Se você não tiver um ponteiro do tipo *va_list**, então especifique o sinalizador *FORMAT_MESSAGE_ARGUMENT_ARRAY* e passe um ponteiro para uma matriz de valores de 32 bits; esses valores são a entrada para a mensagem formatada

como os valores de inserção. Cada inserção precisa ter um elemento correspondente na matriz. A Tabela 1492.2 detalha os sinalizadores de formato para o parâmetro *dwSinaliz*.

Tabela 1492.2 Os valores possíveis para o parâmetro *dwSinaliz*.

Valor	Significado
FORMAT_MESSAGE_ALLOCATE_BUFFER	Especifica que o parâmetro *lpBuffer* é um ponteiro para um ponteiro *PVOID*, e que o parâmetro *nTamanho* especifica o número mínimo de bytes (versão ANSI) ou caracteres (versão Unicode) a alocar para um buffer de mensagem de saída. A função aloca um buffer grande o suficiente para conter a mensagem formatada e coloca um ponteiro no buffer alocado no endereço que *lpBuffer* especifica.
FORMAT_MESSAGE_IGNORE_INSERTS	Especifica que a função precisa ignorar as seqüências de inserção na definição da mensagem e passá-las para o buffer de saída inalteradas. Esse sinalizador é útil para buscar uma mensagem para formatação posterior. Se você definir esse sinalizador, a função ignorará o parâmetro *Argumentos*.
FORMAT_MESSAGE_FROM_STRING	Especifica que *lpOrigem* é um ponteiro para uma definição de mensagem terminada por *NULL*. A definição de mensagem pode conter seqüências de inserção, exatamente como também pode o texto da mensagem em um recurso de tabela de mensagens. Você não pode usar esse sinalizador de formato com *FORMAT_MESSAGE_FROM_HMODULE* ou *FORMAT_MESSAGE_FROM_SYSTEM*.
FORMAT_MESSAGE_FROM_HMODULE	Especifica que *lpOrigem* é um indicativo de módulo que contém o recurso de tabela de mensagens a pesquisar. Se esse indicativo *lpOrigem* for *NULL*, a função pesquisará o arquivo de imagem do aplicativo do processo atual. Você não pode usar esse sinalizador de formato com *FORMAT_MESSAGE_FROM_STRING*.
FORMAT MESSAGE_FROM_SYSTEM	Especifica que a função deve pesquisar a mensagem solicitada no recurso de tabela de mensagens. Se você especificar esse sinalizador com *FORMAT_MESSAGE_FROM_HMODULE*, a função pesquisará a tabela de mensagens do sistema se não encontrar a mensagem no módulo que *lpOrigem* especifica. Você não pode usar esse sinalizador de formato com *FORMAT_MESSAGE_FROM_STRING*.
FORMAT_MESSAGE_ARGUMENT_ARRAY	Especifica que o parâmetro *Argumentos* não é uma estrutura *va_list**, mas, em vez disso, é apenas um ponteiro para uma matriz de valores de 32 bits que representa os argumentos.

Se a função for bem-sucedida, ela retornará o número de bytes (versão ANSI) ou caracteres (versão Unicode) armazenados no buffer de saída, excluindo o caractere *NULL* finalizador. Se a função falhar, ela retornará zero. Para obter informações adicionais sobre erros, chame *GetLastError*.

Dentro do texto da mensagem, a função suporta diversas seqüências de escape para a formatação dinâmica de mensagem. A Tabela 1492.3 mostra essas seqüências de escape e seus significados. Todas as seqüências de escape iniciam com o caractere de porcentagem (%).

Tabela 1492.3 Seqüências de escape possíveis dos caracteres de formatação prefaceados por %.

Seqüência de Escape	Significado
%0	Finaliza uma linha de texto da mensagem sem um caractere preliminar de nova linha. Você pode usar essa seqüência de escape para criar linhas longas ou para finalizar a mensagem sem um caractere de nova linha preliminar. É útil para as mensagens para o usuário.
%n!string de formato printf!	Identifica uma inserção. O valor de *n* pode estar no intervalo de 1 até 99. A string de formato *printf* (que precisa estar delimitada por pontos de exclamação) é opcional, e, por padrão, será *!s!* se você não a especificar. A string de formato *printf* pode conter o especificador * para a precisão ou para o componente largura. Se você especificar * para um componente, a função *FormatMessage* usará a inserção *%n+1*, ela usará *%n+2* se você especificar * para ambos os componentes. A função não suporta os especificadores de formato de ponto flutuante *e*, *E*, *f* e *g* de *printf*. A alternativa é usar a função *sprintf* para formatar o número ponto flutuante em um buffer temporário, depois usar esse buffer como a string de inserção.

A função formata quaisquer outros caracteres que não sejam dígitos que seguem o sinal de porcentagem na mensagem de saída como simplesmente o caractere sem o sinal de porcentagem. A Tabela 1492.4 lista alguns exemplos de saída de caracteres que não são de formatação.

Tabela 1492.4 Exemplo de saída de caracteres não de formatação.

String de Formatação	Saída Resultante
%%	Um único sinal de porcentagem no texto da mensagem formatada.
%n	Uma quebra de linha forçada quando a string de formato ocorre no final de uma linha. Essa string de formato é útil quando *FormatMessage* está fornecendo quebras de linha normais para que a mensagem se acomode em uma determinada largura.
%espaço	Um espaço no texto da mensagem formatada. Você pode usar essa string de formato para garantir o número apropriado de espaços preliminares em uma linha de texto de mensagem.
%.	Um único ponto no texto da mensagem formatada. Você pode usar essa string de formato para incluir um único ponto no início de uma linha sem terminar a definição do texto da mensagem.
%!	Um único ponto de exclamação no texto da mensagem formatada. Você pode usar essa string de formato para incluir um ponto de exclamação imediatamente após uma inserção sem que ela seja confundida com o início de uma string de formato *printf*.

O programa *GenerateError*.cpp, apresentado na dica anterior, utiliza muito a função *FormatMessage*.

E/S Assíncrona com um Objeto Evento do Núcleo 1493

Na Dica 1490, você usou a função *WaitForSingleObject* junto com um indicativo de dispositivo para um dispositivo assíncrono para executar E/S assíncrona. Embora trabalhar com um objeto de dispositivo do núcleo, como detalhou a Dica 1490, seja um tanto simples, não é particularmente útil ao tratar múltiplas solicitações de E/S ao mesmo tempo. Por exemplo, se você estiver tentando executar múltiplas solicitações de E/S assíncronas em um único arquivo simultaneamente, aguardar pelo indicativo de arquivo não ajudará seu processamento, pois ele

se tornará sinalizado quando o primeiro evento terminar, e você precisará esperar novamente para ele liberar — o que poderia resultar em espera interminável.

Você também pode usar a função *CreateEvent* para criar um objeto evento do núcleo. Você pode então identificar esse objeto dentro do membro *hEvento* da estrutura *OVERLAPPED* que você passa para sua função de solicitação de E/S assíncrona (ou *ReadFile* ou *WriteFile*). Quando você passar um evento desse modo, o sistema operacional automaticamente definirá o evento para sinalizado quando a operação de E/S terminar. No entanto, como seu programa pode definir um evento diferente para cada operação de E/S, seus programas podem responder apropriadamente à finalização de uma operação de E/S e não responder a outra.

Toda vez que você executa uma operação de E/S assíncrona, seu programa deve criar um novo evento para essa operação. Desse modo, toda vez que o sistema operacional completar seu processamento, ele definirá o evento para a operação chamadora para um estado sinalizado. Como descreve a dica a seguir, você poderá então esperar pelos eventos que deseja completar.

1494 USANDO WAITFORMULTIPLEOBJECTS COM E/S ASSÍNCRONA

Como você aprendeu, é possível usar *WaitForMultipleObjects* para aguardar um dentre muitos eventos a ocorrer, ou esperar algum subconjunto de muitos eventos a ocorrer. Quando você executa E/S assíncrona, deve usar *WaitForMultipleObjects* para sincronizar seus encadeamentos para um certo conjunto de eventos. Seu programa deve chamar *WaitForMultipleObjects* com os indicativos de todos os eventos que o sistema operacional precisa completar antes que seu programa possa continuar seu processamento, e, depois, esperar aqueles eventos se tornarem um estado sinalizado. Geralmente, você efetuará esse processamento usando código similar ao seguinte fragmento de código:

```
Evento[1] = HANDLE CreateEvent(LPSECURITY_ATTRIBUTES
                    lpAtribEvento, BOOL bResetManual,
                    BOOL bEstadoInicial, LPCTSTR Evento1);
Overlapped1.hEvento = Evento[1];
ReadFile(hArq, bBuffer, sizeof(bBuffer_,
         &dwNumBytesLidos, &Overlapped1);
Evento[2] = HANDLE CreateEvent(LPSECURITY_ATTRIBUTES
            lpAtribEvento, BOOL bResetManual,
            BOOL bEstadoInicial, LPCTSTR Evento2);
Overlapped2.hEvento = Evento[2];
ReadFile(hArq, bBuffer, sizeof(bBuffer_, &dwNumBytesLidos, &Overlapped2);
// processamento adicional aqui
DWORD WaitForMultipleObjects(2, CONST HANDLE *Evento,
                             BOOL bEsperaTodos, INFINITE);
```

O fragmento de código cria um evento e o passa para a primeira ação de leitura. Em seguida, o fragmento de código cria um segundo evento e o passa para a segunda ação de leitura. Finalmente, o código aguarda que ambos os eventos retornem antes de continuar seu processamento.

Os objetos de evento de núcleo são muito úteis para gerenciar a E/S assíncrona. O perigo é se você definir um objeto de evento do núcleo como um evento de auto-reiniciação, porque é possível que um encadeamento trave para sempre enquanto aguarda um evento de auto-reiniciação para reiniciar, embora a função anteriormente tenha completado a operação de E/S de arquivo. Se você chamar *GetOverlappedResult* para determinar quantos bytes a operação de E/S transferiu com sucesso, ela reiniciará o evento para o estado não-sinalizado. Em resumo, observe atentamente a seqüência de funções que você executa ao usar objetos de eventos do núcleo para gerenciar a E/S assíncrona.

Apresentando as Portas de Finalização de E/S 1495

A quarta técnica que seus programas podem usar para efetuar E/S assíncrona é o uso de portas de finalização da E/S. Você geralmente usará as portas de finalização da E/S quando projetar um programa que servirá centenas ou até milhares de usuários (tal como um servidor Web). As portas de finalização de E/S são extremamente seguras e robustas, e podem tratar com segurança os grandes volumes de atividades de comunicação. Quando você criar um aplicativo de serviço, geralmente fará isso de uma entre duas formas:

- No *modelo serial*, um único encadeamento aguarda um cliente para criar uma solicitação (normalmente na rede). Quando a solicitação chega, o encadeamento desperta e trata a solicitação cliente.
- No *modelo concorrente*, um único encadeamento aguarda uma solicitação cliente, e, depois, cria um novo encadeamento para tratar a solicitação. Embora o novo encadeamento trate a solicitação cliente, o encadeamento original retorna no laço e aguarda outra solicitação cliente. Quando o encadeamento que está tratando a solicitação cliente completa seu processamento, o encadeamento morre.

O modelo serial é muito limitado, pois não trata bem múltiplas solicitações simultâneas (porque apenas um único encadeamento trata as solicitações). Em contraste, o modelo concorrente é capaz de tratar um número extremamente grande de solicitações simultaneamente, porque cada solicitação recebe seu próprio encadeamento. Quando você projetar serviços do Windows NT, seus programas geralmente usarão o modelo concorrente. Você somente usará as portas de finalização de E/S em conjunção com aplicativos que usam o modelo concorrente.

É óbvio que, a criação de um serviço de modelo concorrente está bem além da abrangência deste livro. É suficiente para você compreender a diferença entre serviços do modelo serial e serviços do modelo concorrente à medida que continuar a desenvolver seus programas.

Nota: Você pode usar as portas de finalização de E/S somente sob o Windows NT. O Windows 95 não tem a funcionalidade necessária para implementar as portas de finalização de E/S.

Usando E/S Alertável para o Processamento Assíncrono 1496

Como você aprendeu, sempre que uma função cria um encadeamento, o sistema também cria uma fila de mensagem para esse encadeamento e o associa com o encadeamento. O sistema operacional também cria outra fila para esse encadeamento, conhecido como fila *Asynchronous Procedure Call* (APC, ou Chamada de Procedimento Assíncrono). O sistema operacional usa as funções de baixo nível dentro do núcleo para criar e manter a fila APC. Como o sistema operacional usa funções de baixo nível do núcleo para manter a fila APC, essa fila é um método muito rápido e eficiente para gerenciar a E/S assíncrona.

É possível para seus programas fazerem solicitações de E/S e ter funções que enviam os resultados das solicitações de E/S diretamente para a fila APC do encadeamento chamador. Para enviar solicitações de E/S completadas para a sua fila APC, você usará as funções *ReadFileEx* e *WriteFileEx*, como mostrado aqui:

```
BOOL ReadFileEx(HANDLE hArq, LPVOID lpBuffer,
       DWORD nNumBytesALer, LPOVERLAPPED lpSobreposta,
       LPOVERLAPPED_COMPLETION_ROUTINE lpRotinaFinaliz);
BOOL WriteFileEx(HANDLE hArq, LPCVOID lpBuffer,
       DWORD nNumBytesAGravar, LPOVERLAPPED epSobreposta, LOVERLAPPED_
COMPETION_ROUTINE_lpRotinaFinaliz);
```

Como você pode ver, ambas as funções aceitam como seu último parâmetro o endereço de uma rotina de finalização para executar quando eles completarem seu processamento. A Dica 1498, à frente, explicará *ReadFileExe* e *WriteFileEx* em maiores detalhes. Você precisa usar o seguinte protótipo para a rotina de finalização que ambas as funções usam:

```
VOID WINAPI FileIOCompletionRoutine(
   DWORD dwCdgErro,                // código de finalização
   DWORD dwNumBytesTransferidos, // número de bytes transferidos
   LPOVERLAPPED lpSobreposta    // ponteiro para estrutura
```

```
                                    // com informações de E/S
);
```

O parâmetro *dwCdgErro* especifica o status de finalização da E/S. O parâmetro *dwCdgErro* pode ser um dos valores mostrados na Tabela 1496.

Tabela 1496 Valores possíveis para o parâmetro ***dwCdgErro***.

Valor	Significado
0	A E/S foi bem-sucedida.
ERROR_HANDLE_EOF	A função tentou ler além do fim do arquivo.

O parâmetro *dwNumBytesTransferidos* especifica o número de bytes transferidos. Se um erro ocorrer, esse parâmetro será zero. O parâmetro *lpSobreposta* aponta para a estrutura *OVERLAPPED* que a função de E/S assíncrona especifica. O Windows não usa o membro *hEvento* da estrutura *OVERLAPPED*; o aplicativo chamador poderá usar esse membro para passar informações para a rotina de finalização. O Windows não usa a estrutura *OVERLAPPED* após o programa chamar a rotina de finalização, de modo que a rotina de finalização pode desalocar a memória usada pela estrutura *OVERLAPPED*.

A função *FileIOCompletionRoutine* é um marcador de lugar para um nome de função definido pelo aplicativo ou definido pela biblioteca. Retornar da função *FileIOCompletionRoutine* permite que o Windows chame outra rotina de finalização de E/S. Todas as rotinas de finalização em espera são chamadas antes que a espera do encadeamento alertável seja satisfeita com um código de retorno de *WAIT_IO_COMPLETION*. O Windows poderá chamar as rotinas de finalização que estão em espera em qualquer ordem. Ele pode ou não chamar as rotinas na ordem em que o programa completa as funções de E/S. Toda vez que o Windows chama uma rotina de finalização, ele usa parte da pilha do aplicativo. Se a rotina de finalização fizer E/S assíncronas adicionais e esperas alertáveis, a pilha poderá crescer.

1497 A E/S Alertável Somente Funciona no Windows NT

A E/S alertável é uma técnica avançada para tratar E/S assíncronas que usam a fila de mensagem de E/S e uma ou mais funções de callback. Como a E/S alertável usa as versões estendidas das funções *ReadFile* e *WriteFile*, seu programa somente pode usar E/S alertável se tiver certeza de que somente rodará em sistemas Windows NT. Se você tentar usar *ReadFileEx* ou *WriteFileEx* em um sistema Windows 95 ou Win32s, ambas as funções retornarão *False*, e não executarão nenhum processamento. Uma chamada à *GetLastError* retornará *ERROR_CALL_NOT_IMPLEMENTED*. Não tente usar E/S alertável em uma máquina Windows 95, pois ela terá efeitos imprevisíveis.

1498 Usando ReadFileEx e WriteFileEx

Como você aprendeu na Dica 1494, seus programas podem usar as funções *ReadFileEx* e *WriteFileEx* para executar E/S assíncrona em arquivo sob o Windows NT. A função *ReadFileEx* lê dados de um arquivo assincronamente. Os programadores projetaram a função *ReadFileEx* unicamente para a operação assíncrona, ao contrário de *ReadFile*, que os programadores projetaram tanto para a operação síncrona quanto assíncrona. *ReadFileEx* permite que um aplicativo execute outro processamento durante uma operação de leitura de arquivo. A função *ReadFileEx* informa seu status de finalização assincronamente, chamando uma rotina de finalização que você especifica quando completa a leitura, e quando o encadeamento chamador está em um estado de espera alertável. Você usará a função *ReadFileEx* dentro de seus programas, como segue:

```
BOOL ReadFileEx(
   HANDLE hArq,              // indicativo do arquivo a ler
   LPVOID lpBuffer,          // endereço do buffer
   DWORD  nNumBytesALer,     // número de bytes a ler
   LPOVERLAPPED lpSobreposta, // endereço do deslocamento
   LPOVERLAPPED_COMPLETION_ROUTINE
       lpRotinaFinalizacao    // endereço da rotina de finalização
);
```

A função *ReadFileEx* aceita os parâmetros detalhados na Tabela 1498.

Tabela 1498 Os parâmetros para a função ***ReadFileEx***.

Parâmetro	Descrição
hArq	Um indicativo de abertura que especifica a entidade arquivo a partir da qual ler. Você precisa criar esse indicativo de arquivo com o sinalizador *FILE_FLAG_OVERLAPPED*, e precisa ter o acesso de arquivo *GENERIC_READ*. O parâmetro *hArq* pode ser qualquer indicativo que a função *CreateFile* abriu com o sinalizador *FILE_FLAG_OVERLAPPED*.
lpBuffer	Aponta para um buffer que recebe os dados que a função lê do arquivo. O aplicativo não deve usar esse buffer até que a função complete a operação de leitura.
nNumBytesALer	Especifica o número de bytes para a função ler do arquivo.
lpSobreposta	Aponta para uma estrutura de dados *OVERLAPPED* que fornece dados para a função usar durante a operação de leitura assíncrona (sobreposta) de arquivo. Se o arquivo que *hArq* especificou suporta deslocamento de byte, o processo chamador de *ReadFileEx* precisa especificar um byte de deslocamento dentro do arquivo onde a leitura deve iniciar. O processo chamador especifica o byte de deslocamento definindo os membros *Offset* e *OffsetHigh* da estrutura *OVERLAPPED*. Se a entidade arquivo que *hArq* especificou não suportar deslocamentos de byte (por exemplo, se for uma canalização nomeada), o processo chamador precisará definir os membros *Offset* e *OffsetHigh* como zero, ou *ReadFileEx* falhará. A função *ReadFileEx* sinaliza a finalização de sua operação de leitura chamando — ou enfileirando uma chamada à — rotina de finalização apontada por *lpRotinaFinalizacao*, de modo que ela não precisa de um indicativo de evento. A função *ReadFileEx* usa os membros *Internal* e *InternalHigh* da estrutura *OVERLAPPED*. Um aplicativo não deve definir esses membros. A estrutura de dados *OVERLAPPED* para a qual *lpSobreposta* aponta precisa permanecer válido durante a duração da operação de leitura.
lpRotinaFinalizacao	Aponta para a rotina de finalização para o Windows chamar quando a operação de leitura estiver completa e o encadeamento chamador estiver em um estado de espera alertável.

Se a função *ReadFileEx* for bem-sucedida, ela retornará um valor diferente de zero. Se falhar, ela retornará zero. Para obter informações adicionais de erro, chame *GetLastError*. Se a função for bem-sucedida, o encadeamento chamador terá uma operação de E/S assíncrona pendente: a operação de leitura sobrecarregada do arquivo. Quando a operação de leitura sobrecarregada terminar e o sistema bloquear o encadeamento chamador em um estado de espera alertável, o sistema chamará a função para a qual *lpRotinaFinalizacao* aponta, e o estado de espera terminará com um código de retorno de *WAIT_IO_COMPLETION*.

Se a função for bem-sucedida e a operação de leitura de arquivo terminar, mas o encadeamento chamador não estiver em um estado de espera alertável, o sistema enfileirará a chamada da rotina de finalização, retendo a chamada até que o encadeamento chamador entre em um estado de espera alertável. Se *ReadFileEx* tentar ler além do final do arquivo, a função retornará zero, e *GetLastError* retornará *ERROR_HANDLE_EOF*.

Se outro processo bloquear uma porção do arquivo que *hArq* especifica e a operação de leitura especificada em uma chamada à *ReadFileEx* sobrepuser a porção bloqueada, a chamada à *ReadFileEx* falhará. Se *ReadFileEx* tentar ler dados a partir de um mailslot cujo buffer é pequeno demais, a função retornará *False*, e *GetLastError* retornará *ERROR_INSUFFICIENT_BUFFER*. Os aplicativos não deverão ler nem escrever no buffer de entrada que uma operação de leitura estiver usando até que a operação de leitura termine. Um acesso prematuro ao buffer de entrada pode levar à corrupção dos dados que a função lê nesse buffer. A função *ReadFileEx* pode falhar se existem excessivas solicitações de E/S pendentes. No caso desse tipo de falha, *GetLastError* poderá retornar *ERROR_USER_BUFFER* ou *ERROR_NOT_ENOUGH_MEMORY* (como você aprendeu na Dica 1488).

Se você tentar ler uma unidade que não tem um disquete, o sistema exibirá uma caixa de mensagem pedindo que o usuário repita a operação. Para evitar que o sistema exiba essa caixa de mensagem, chame a função *SetErrorMode* com *SEM_NOOPENFILEERRORBOX*. Se *hArq* for um indicativo para uma canalização nomeada

ou outra entidade arquivo que não suporte o conceito de deslocamento de byte, os membros *Offset* e *OffsetHigh* da estrutura *OVERLAPPED* para a qual *lpSobreposta* precisará ser zero, ou *ReadFileEx* falhará.

1499 USANDO UMA ROTINA DE CALLBACK DE FINALIZAÇÃO

Como você aprendeu na Dica 1497, seus programas precisam usar uma rotina de callback de finalização com *ReadFileEx* e *WriteFileEx*. Você precisa usar o seguinte protótipo para a rotina de finalização que ambas as funções chamam:

```
VOID WINAPI FileIOCompletionRoutine(
   DWORD dwCdgErro,              // código de erro
   DWORD dwNumBytesTransferidos,// número de bytes transferidos
   LPOVERLAPPED lpSobreposta    // ponteiro para estrutura
                               // com informações de E/S
);
```

Quando você chamar um dos objetos *Wait* e colocar o encadeamento em um estado alertável, o sistema operacional primeiro verificará a fila APC (Chamada de Procedimento Assíncrono) do seu encadeamento. Se pelo menos um item estiver na fila, o sistema não colocará o encadeamento para dormir; em vez disso, ele retirará o item da fila APC e seu encadeamento chamará a rotina de callback, passando-lhe o código de erro da solicitação da E/S completada, o número de bytes transferidos e o endereço da estrutura *OVERLAPPED* que o encadeamento originalmente passou na solicitação de E/S. Após a rotina de callback efetuar seu processamento, o sistema novamente verifica se existirem mais itens na fila APC. Se existirem mais itens, o sistema os passará para a rotina de callback na ordem. Se não existirem mais itens, a função alertável retornará e o encadeamento continuará seu processamento sem adormecer. Portanto, a única hora em que seu encadeamento adormece é quando não existem itens na fila APC.

1500 USANDO UM PROGRAMA DE E/S ALERTÁVEL

Como você aprendeu, seus programas podem usar as poderosas técnicas de E/S alertáveis para executar processamento robusto da entrada e saída assíncrona de arquivo. O CD-ROM que acompanha este livro inclui os programas *Alertable_IO.cpp*, que usa os conceitos da E/S alertável para efetuar uma simples tarefa de cópia. Quando você compilar e executar o programa, ele usará a E/S alertável para preparar e copiar um arquivo, e o alertará para seu processamento e finalização.

Quando o programa inicia, ele cria um conjunto de solicitações de E/S. Para fazer isso, ele inicializa um conjunto de estruturas *MAX_PENDING_IO_REQS* que ele usará para informar o sistema operacional do maior número possível de solicitações simultâneas de E/S. Cada estrutura contém uma estrutura *OVERLAPPED*, embora nenhuma contenha um ponteiro dentro do membro *hEvento*. Além da estrutura *OVERLAPPED* que cada solicitação de E/S requer, cada solicitação requer um buffer de memória, que o programa também mantém dentro da estrutura *IO_REQS*.

Após o programa inicializar uma estrutura *IO_REQS*, ele chama a função *ReadFileEx* para emitir a solicitação para o arquivo de leitura a partir do sistema operacional. Nesse ponto, o programa começa a usar E/S alertável. O processo retorna imediatamente do sistema operacional. Nesse ponto, o programa começa a usar E/S alertável. O programa imediatamente retorna para a mensagem da caixa de diálogo, e o usuário pode imediatamente especificar outro arquivo para copiar. No entanto, no segundo plano, *ReadFileEx* está encontrando e lendo o arquivo. Quando ela termina, altera o processo (por meio da rotina de callback), que usa as informações que ela anteriormente leu para gravar a cópia do arquivo. Embora o programa seja grande demais para listar aqui por inteiro, vale a pena analisar as duas funções de callback:

```
void WINAPI WriteCompletionRoutine(DWORD dwCdgErro,
   DWORD dwNumBytesTransf, LPOVERLAPPED pSobreposta);

void WINAPI ReadCompletionRoutine(DWORD dwCdgErro,
   DWORD dwNumBytesTransf, LPOVERLAPPED pSobreposta)

 {
   PIOREQ pIOReq = (PIOREQ) pSobreposta;
   chASSERT(dwCdgErro == NO_ERROR);
```

```
   g_cs.nReadInProgress--;

// Arredonda para cima o número de bytes a gravar em um
// limite de setor

   dwNumBytesTransf=(dwNumBytesTransf + g_cs.dwPageSize-1) &
       ~(g_cs.dwPageSize-1);
   chVERIFY(WriteFileEx(g_cs.hFileDst, pIOReq->pbData,
dwNumBytesTransf,
pSobreposta,
         WriteCompletionRoutine));
   g_cs.nWriteInProgress++;
 }

void WINAPI WriteCompletionRoutine(DWORD dwCdgErro, DWORD
dwNumBytesTransf,
       LPOVERLAPPED pSobreposta);
 {
   PIOREQ pIOReq = (PIOREQ) pSobreposta;
   chASSERT(dwCdgErro == NO_ERROR);
   g_cs.nWritesInProgress--;

   if (g_cs.ulNextReadOffset.Quadpart < g_cs.ulFileSize.Quadpart)
   {
      // a função ainda não leu além do final do arquivo
      // portanto, lê o próximo bloco de dados
      pSobreposta->Offset = g_cs.ulNextReadOffset.LowPart;
      pSobreposta->OffsetHigh = g_cs.ulNextReadOffset.HighPart;
      chVERIFY(ReadFileEx(g_cs.hFileSrc, pIOReq->pbData,
            BUFFSIZE, pSobrecarregada, ReadCompletionRoutine));
      g_cs.nReadsInProgress++;
      g_cs.ulNextReadOffset.Quadpart += BUFFSIZE;
   }
 }
```

Como você pode ver, a função de callback *ReadFileCompletionRoutine* chama a função *WriteFileEx* com as informações que *ReadFileEx* retorna e o endereço da função de callback *WriteFileCompletionRoutine*. A função *WriteFileCompletionRoutine* confere o tamanho atual da cópia cada vez que *WriteFileEx* retorna; se a cópia não estiver completa, *WriteCompletionRoutine* mudará a localização de deslocamento dentro do arquivo e chamará *ReadFileEx* novamente. Se a cópia estiver completa, a função terminará e o programa tratará a limpeza de arquivo em outra parte. Como você pode ver, implementar E/S alertável dentro de seus programas é um modo relativamente simples e útil de executar a E/S assíncrona.

Nota: O programa Alertable_IO.cpp apenas rodará corretamente em sistemas Windows NT.

Índice Analítico

Os números deste Índice referem-se ao número da dica.

#define, diretiva, usando para criar uma constante, 132
#else, diretiva, usando a, 150, 152
#error, diretiva, usando para exibir mensagem de erro, 138
#if, diretiva, usando, 151
#ifdef, diretiva, usando, 149
#ifndef, diretiva, usando para criar um arquivo de cabeçalho, 147
#include, diretiva, usando 149
#include, comando, definição, 5
#line, diretiva usando para modificar o número de linha atual, 137
#undef, diretiva, usando, 143
&, operador
  aplicando a uma matriz, 510
  compreendendo, 88
  usando, para determinar o endereço de uma variável, 229, 508
(), operador, sobrecarregando o, 960
*, operador, usando para desreferenciar o valor de um ponteiro, 512
+, operador, sobrecarregando o, 947
, operador
  compreendendendo o, 962
  sobrecarregando, 962
  operador, sobrecarregando o, 948
-> operador, sobrecarregando o, 961
LIB, arquivos, compreendendo, 690
OBJ, arquivos, compreendendo, 691
/ comentário */* usando para comentar, 16
//, usando para comentar, 16
::, operador
  compreendendo o, 894
  revisando o, 908
;, operador, compreendendo, 22
<< operador de deslocamento, 94
== operador, sobrecarregando o, 1197
>> operador de deslocamento, 94
?, operador, compreendendo, 92
[] operador, sobrecarregando o, 959
\n (caractere de nova linha), usando o, *7*
^, operador, compreendendo, 89
bios_equiplist, função, usando, 563
_bios_keybrd, função, usando, 562
_bios_serialcom, função, usando, 564
_cexit, função, usando, 774
chain interrupt, usando, 768
_chdrive, *função que seleciona a unidade de disco atual*, 351
_chmod, função para controlar os atributos de arquivo, 380
_cplusplus, constante, usando a, 872, 142
*DATE_*, constante de pré-processador, usando a, 140
_dos_getdiskfree, função, usando para selecionar a unidade de disco atual, 352
dos_getdrive, função, usando para determinar a unidade de disco atual, 350
_dos_getfileattr, usando para controlar os atributos de arquivo, 380
_dos_getftime, função, usando para obter a data e a hora de um arquivo, 411
dos_getvect, usando a, 764
_dos_setdrive, função, usando para selecionar a unidade de disco atual, 351
_dos_setfileattr, função, usando para controlar os atributos do arquivo, 380
_dos_setftime, função, usando para definir a data e a hora de um arquivo, 413
_dos_setvect, função, usando a, 765
fastcall, modificador, 780
  regras para a passagem de parâmetros, 781
*_FILE_*, constante do pré-processador, usando, 135
_fsopen, função, usando para abrir um canal de arquivo compartilhado, 437
_fullpath, função, usando para criar um nome de arquivo completo, 399
*_LINE_*, constante do pré-processador, usando a, 135
_makepath, função, usando para criar um nome de caminho completo, 401

Aguardando uma digitação, 648
Amarração precoce e tardia
  compreendendo, 1097
  definição, *1086*
  escolhendo entre, 1098
  exemplo de, 1099
Ambiente,
  como o DOS trata o, 684
  compreendendo, 676
  DOS, acrescentando elementos a, 687
Ambiguidade da classe, evitando com classes virtuais, 1068, 1069
Amigas
  acesso, restringindo, 1046
  mútuas, definição, 1070
  usando para conversões, 1107
Apelidando os ponteiros, definição, 251
API do Windows, apresentação, 1263
Área de memória alta, compreendendo, 583
Área cliente, definição, *1255*
Argumentos da função, padrão,
  evitando erros com, 1142
  usando, 1141
  versus sobrecarga de função, 1143
Argumentos da linha de comando,
  compreendendo, 669
  delimitados, trabalhando com, 672
  exibindo o, 671
  exibindo um contador de, 670
  usando, para exibir o conteúdo do arquivo, 673
Aritmética de ponteiros, compreendendo, 515
Armazenamento estático, compreendendo, 978
Arquivo de cabeçalho,
  compreendendo, 13
  criando, usando a diretiva *#include*, 147
  ctype.h,
    _tolower, macro, compreendendo, 211
    _toupper, macro, compreendendo, 210
  definição, 5
  iostream.h, compreendendo, 820
  localizando, ajudando o compilador, 14
  stdlib.h,
    max, macro, usando, 341
    min, macro, usando, 341
  strings.h, usando o, *1177*
Arquivo de paginação, definição, 1360
Arquivo de recurso,
  compreendendo, 1258, 1312
  definição, 1257
Arquivo de resposta, definição, 701
Arquivo mapeando na memória,
  definição, 1380
  usando para compartilhar dados, 1380
Arquivo mapeando, abrindo, 1462
Arquivo(s)

abrindo um, 402
para acesso compartilhado, 426
usando a função *CreateFile*, 1452
usando, função *fopen*, 359
acessando, usando os serviços de arquivo do DOS, 440
buffer,
alocando um, 419
atribuindo um, 418
builtins.mak, usando o, 717
cabeçalho,
compreendendo, 13
criando, usando a diretiva *#include*, 147
definição, 5
conteúdo, bloqueando, 428, 429
copiando, usando a função *CopyFile*, 1471
criando um, 403, 439
em um diretório específico, 438
data da criação
definindo uma, 413
para a atual, 414
obtendo, 411
excluindo um, 377
de um diretório, 1473
fechando um, 402
todos abertos, 384
usando a função *CloseHandle*, 1458
usando a função *fclose*, 361
gravando dados,
usando a função *ReadFile*, 1457
usando a função *WriteFile*, 1456
gravando um, 403
hora da criação
obtendo, 411, 1466
definindo a, 413
com a hora atual, 414
inclusão, definição, 5
lendo um, 404
MAKE,
colocando múltiplas dependências em um, 706
comentando, 705
criando um simples, 703
incluindo um segundo, 713
terminando, com um erro, 715
mapeado na memória, usando, para compartilhar dados, 1380
mapeando, abrindo, 1462
mapeando para a memória virtual, 1460
modo de acesso, definição, 379
movendo, 1472
renomeando, 376, 1472
tamanho,
determinando 382
mudando, 416
obtendo, 1465
temporário,
criando, 1480
removendo, 389
testando o fim do arquivo com a função feof, 447
Arquivo-fonte, ASCII, criando, 2
Arquivos abertos, fechando todos, 384
Arquivos binários, copiando, 1008
Arquivos LIB, compreendendo, 690
ASCII para numérico, função, 188
ASCII, representação numérica, convertendo uma, 198
Asin, função, usando a, 323
Asm, palavra-chave de C++, usando, 854
Aspas, simples *versus* duplas, 499
Associatividade de operador, compreendendo, 83
Asynchronous Procedure Cal. (APC), fila, 1495
Asin, função, usando a, 324
Atexit, função, usando a 689
Atributo, virtual herdando, 1091
Atributos de arquivo,
compreendendo, 1463
controlando, 380
mudando, 1464
obtendo, 1464
Auto, palavra-chave, compreendendo, 276
Avaliação de curto-circuito, definição, 837
Avaliação hesitante, definição, 837
Average_value, função, usando, 217

Barra de título, definição, 1255
Barras de menu, definição, 1255
Barras de rolagem da área não-cliente, compreendendo, 1347
Barras de rolagem da área cliente, definição, 1347
Barras de rolagem,
apresentando, 1346
área não-cliente, definição, 1347
área cliente, definição, 1347
definição, 1255
definições, obtendo o atual, 1351
elevador, definição, 1346
intervalo, compreendendo, 1349
mensagens, compreendendo, 1350
posição, compreendendo, 1349
tipos, compreendendo os diferentes, 1347
Barras invertidas, compreendendo o uso nos nomes de diretório, 394
Base de E/S, definindo a, 833
Bdos, função usando, 565
Biblioteca de execução, compreendendo, 261
Bibliotecário, definição, 690
Bibliotecas,
classe
canal antigo, compreendendo, 1002
compreendendo, 967
compreendendo, 690
criando uma, 693
operações, comuns, tabela de, 694
rotinas, listando as, 695
usando, para reduzir o tempo de compilação, 696
BIOS (Serviços Básicos de Entrada e Saída) compreendendo, 550
Biosdisk, função, usando para
efetuar operação de E/S no disco, 357
testar a disponibilidade da unidade de disco, 358
Biosmemory, função, usando, 567
Biosprint, função. usando para acessar a impressora, 554
Biostime, função, usando, 629
BitBlt, função, 1435
BITMAP, palavra-chave, usando com um arquivo de recurso, 1433
BITMAPINFO, estrutura, 1433
BITMAPINFOHEADER, 1433
Bkwdlist, função, classe *linked_list*, compreendendo a, 1188
Bloco *catch*, compreendendo quando o programa executa, 1133
Bloco de memória virtual, alocando um 1371
Bloco de memória,
alocado, alterando o tamanho, 601
virtual, alocando um, 1371
Bool, tipo de dado
apresentando, 1161
usando, 1162
Botão do mouse,
clique duplo, definição, 1343
lendo o, 1338
permutando, 1344
respondendo ao, 1336
Break, comando, usando para finalizar um laço, 127
Brk, função, compreendendo, 602
Bsearch, função, usando para pesquisar uma matriz Classificada, 504

Buffer, arquivo,
alocando um 419
atribuindo um, 418
Builtins.mak, arquivo, usando, 717
BUTTON, classe do Windows, definição, 1276

C *versus* C++, testando, 142
C++
apresentando, 803
compilação, determinando, 872
herança em, 1048
programa, criando um simples, 805
referências, compreendendo, 847
tabelas de novas palavras-chave, 822
Cabs, função, usando, 325
Caixa
convertendo uma string, 175
em C, compreendendo, 8
maiúscula, convertendo uma string para, usando a Função *strupr*, 175
minúscula, convertendo uma string para, usando a Função *strlwr*, 175
Caixa de diálogo modal e não-modal, definição, 1320
Caixa de diálogo
compreendendo, 1319
controle, definindo o, 1325
exibindo, usando a macro *DialogBox*, 1326
fechando a, 1334
gabarito,
componentes de, compreendendo o, 1322
criando um, 1323
laço de mensagem, compreendendo, 1327
modal do sistema, definição, 1320
modal e não-modal, definição, 1320
processamento padrão de mensagem dentro, 1331
tipos, definindo, 1320
Caixas de lista, respondendo às seleções do usuário dentro, 1333
CallNamedPipe, função para conexão a uma canalização nomeada, 1483
Calloc, função, usando para alocar memória, 596
Caminho temporário, obtendo um usando a função *GetTempPath*, 1479
Campos de bit, compreendendo, 485
Canal de arquivo,
abrindo um, 1003
compartilhado, abrindo um 437
compreendendo, 365
erro, testando, 381
fechando um 1004
operações, combinando, 1007
ponteiro, controlando o, 1021
reabrindo, usando a função *freopen*, 452
Canal de E/S,
C++, compreendendo, 980
esvaziando um, 383
operações, sincronizando com a função *stdio*, 979
status, testando, usando a função-membro *rdstate*, 1024
Canal de entrada,
compreendendo, 092
definição, 982
Canal de saída, compreendendo, 981
Canal de saída, manipulador, definindo um, 110
Canal,
arquivo,
abrindo um, 1003
fechando um, 1004
cerr E/S, escrevendo saída para o, 812
E/S *clog* executando saída com, 817
E/S, C++, compreendendo, 980
entrada, C++, compreendendo, 982
saída,
C++, compreendendo, 981
definindo um manipulador para, 1100
string de caracteres, compreendendo, 1026
Canalização nomeada,
chamando uma, 1483
conectando uma, 1482
criando uma, 1481
desconectando uma, 1484
Caractere de controle, definição, 202
Caractere de nova linha (\n), usando, 7
Caractere de preenchimento, canal de E/S *cout*, 828
Caractere(s)
ASCII, trabalhando com, 212
atribuindo, 52
convertendo
para maiúsculas, 210
para minúsculas, 211
copiando de uma string para outra, 168
determinando,
alfanumérica, usando a macro *isalnum*, 199
caixa 205
controle, 202
dígito, 203
do alfabeto, usando a macro isalnum, 199
espaço em branco, 208
gráfico, 204
imprimível, 206
símbolo da pontuação, 207
valor ASCII, 201
valor hexadecimal, 209
escape,
definidos por C, 52, 74
trabalhando com C, 74
escrevendo na tela usando a macro *putchar*, 287
exibidos por printf, determinando o número, 75
individual, usando o canal de E/S *cin*, 855
lendo
do teclado,
usando a função *getche*, 292
usando a macro *getchar*, 286
localizando
primeira ocorrência em uma string, 176
última ocorrência em uma string, 178
redirecionados, exibindo um contador de, 661
Caracteres redirecionados, exibindo um contador de, 661
Catch, comando
múltiplo, usando com um único bloco *try*, 1134
compreendendo o, 1130
Cdecl, modificadores, compreendendo, 801
Cdecl, palavra-chave compreendendo, 239
Ceil, função usando a, 326
Cerr, canal de E/S, escrevendo saída no, 81
Cgets, função, usando para ler uma string de caracteres do Teclado, 802
Chamada de função,
definição, 252
rápida, compreendendo, 780
Chamada de Procedimento Assíncrono (APC), fila, 1495
Chamada por referência
compreendendo, 228
usando para alterar o valor de um parâmetro, 231
Chamada por valor, compreendendo, 226
Chamadas rápidas de função, compreendendo, 780
Change_first, função para modificar parâmetros específicos, 232
Char, tipo
valores, exibindo com *printf*, 59
variável, compreendendo, 33

Char, tipo de variável, compreendendo, 33
Charcnt, função, usando para contar o número de ocorrências do caractere, 182
Chdir, função, usando para mudar o diretório atual, 395
Chmod, função, usando para definir o modo de acesso do Arquivo, 379
Chsize, função, usando para alterar o tamanho do arquivo, 416
Cin, canal de E/S
compreendendo, 814-816
lendo caracteres individuais com, 855
manipulador *ws*, 966
recebendo entrada com, 813
usando, para ler uma linha, 868
Cin.get, usando, 868
Cin.getline,
em um laço, 869
para redirecionar a entrada, 869
usando, 868
Classe-base,
compreendendo, 1049
construtoras, passando parâmetros para, 1065
declarando como privadas, 1059
Classe de lista ligada dupla, genérica, criando uma, 1191
Classe de matriz genérica, criando e compreendendo, 1125
Classe derivada,
compreendendo, 1049
definição, 1049
usando declarações de acesso dentro, 1066, 1067
Classe genérica,
compreendendo, 1121
criando, com dois tipos de dados genéricos, 1123
definição, 1001
usando, 1122
Classe local, compreendendo, 941
Classes amigas, definição, 920
Classes de base virtuais, usando para evitar ambigüidade, 1068
Classes de E/S, baseadas em matrizes, compreendendo, 1025
Classes,
abstrata, compreendendo, 1095
amigas, definição, 920
atribuindo um tipo para outra classe, 1106
base,
compreendendo, 1049
definida, 1049
bibliotecas, compreendendo, 967
C++, compreendendo, 886, 917
componentes, definindo a, 892
compreendendo, 885
definidas, 818
derivadas,
compreendendo, 1049
definidas, 1049
derivando uma, 1050
duplicar, usando gabaritos para eliminar, 1120
em ninhos, compreendendo, 940
escopo, compreendendo, 939
genéricas,
com uma estrutura, 1196
compreendendo, 1121
criando, com dois tipos de dados genéricos, 1123
definição, 1001
matriz, criando e compreendendo uma, 1125
usando, 1122
com uma lista *char* 1194
com uma lista *double*, 1195
local, compreendendo, 941
membros,
acessando, 907
estática, compreendendo, 911
recursiva, 1083
modelo,
criando um simples, 891
implementando um simples, 891
nomes,
omitindo, em declarações, 895
usando, em declarações, 895
type_info, compreendendo, 1157
valores, inicializando, 909, 910
variáveis, matriz de, criando uma, 943
versus estruturas, quando usar, 890
Classificando, problemas com, 500
Clearerr, macro, usando para testar erros no canal de Arquivos, 381
Clique duplo, botão do mouse, definição, 1343
Clock, função, usando, 625
Clog, canal de E/S, executando saída com, 817
Close, função, usando para fechar um arquivo, 817
Close, membro, usando para fechar um canal de arquivo, 1004
CloseHandle, função
para fechar arquivo, 1458
usando, 1377
Clreol, função, usando para apagar final da linha atual, 305
Clrscr, função, usando para limpar a tela, 304
CMOS, compreendendo, 612
Co-processador matemático, comandos do, 800
Co-processador matemático, determinando se está presente, 721
Código
compactação, compreendendo, 784
invariante, compreendendo, 782
Código de máquina, definição, 1
Código *in-line*, usando, 778
Código invariante, compreendendo, 782
Códigos de teclas virtuais, tabela de, 1340
Colunas, matriz, compreendendo, 469
Comando
DELTREE
criando seu próprio, 398
usando, para remover uma árvore de diretório, 398
DOS BREAK, usando, 555
interno do DOS, chamando um, 730
MORE,
criando, 659
temporizado, criando, 662
Comandos compostos,
compreendendo, 100
definição, 100
Comandos simples, compreendendo, 100
COMBOBOX, classe, Windows, definição, 1276
Comentários,
definição, 16
usando para excluir comandos do programa, 20
Compact, modelo de memória, compreendendo, 617
Compactação
código, compreendendo, 784
laço, compreendendo, 785
Compactação do laço, compreendendo, 795
Compartilhamento de arquivo, compreendendo, 426
Compilações, agilizando, 15
Compilador,
advertências,
controlando, 19
compreendendo, 18
ajudando a localizar os arquivos de cabeçalho, 14
definição, 1
Compilando, definição, 5
Comprimento da lista, escrevendo um programa para determinar, 1200
Concatenação, definição, 169
Concatenando strings,

definição, 169
sobrecarregando o operador de atribuição (+), 1169
Condição, testando usando o comando *if*, 99
CONFIG SYS, linha FILES=, compreendendo, 367
Conflitos de nomes,
ao usar amigas, 1047
classes-base *versus* derivadas, 1061
membro e parâmetro, resolvendo, 942
resolvendo, classe e base, 1062
Conflitos de nomes, resolvendo, 263
ConnectNamedPipe, função, usando, 1482
Const, modificador, usando em declarações de variáveis, 732
Const, palavra-chave,
compreendendo o uso, 250
usando em C++, 838
Const_cast, operador, usando o, 1148
Constante string, declarando, usando um ponteiro, 526
Constante,
anulando a definição de uma, 143
comparando, 144
criando, usando a diretiva *#define* para, 132
definição, 132
nomeando, 134
pré-processador,
*DATE_*, usando, 140
*_FILE_*, usando, 135
*_LINE_*, usando, 135
*_TIME_*, usando, 140
string, declaração, usando um ponteiro, 526
usando para definir matrizes, 460
Constante, _cplusplus, usando, 872
para determinar o modo do compilador, 142
Constante, *_STDC_*, usando para testar a aderência ao padrão ANSI, 141
Constantes do programa, definição, 132
Construtora de classe, herdada, exemplo, 1064
Construtora parametrizada, definição, 922
Construtoras-base, compreendendo, 1051
Construtoras derivadas, compreendendo, 1051
Construtoras,
base, compreendendo, 1051
classe *Strings*, escrevendo a, 1166
classes herdadas, exemplo, 1064
convertendo dados em uma, 1105
derivadas, compreendendo, 1051
ordem, compreendendo, 1058
Contenção de mutex, definição, 1411
Conteúdo da janela, rolando, 1352
CONTEXT, estrutura, 1386
Contexto do encadeamento, definição, 1386
Continue, comando, compreendendo o, 126
Control+break, compreendendo, 555
Controles,
caixa de diálogo, definindo a, 1325
definição, 1276
Conversão, padrão, compreendendo, 788
Conversão, usando amigas para, 1107
Conversões de classe, compreendendo, 1104
Cópia bit a bit, definição, 937
Co-processador matemático,
comandos, 800
determinando se está presente, 721
CopyFile, função, usando para copiar arquivos, 1471
Cor de fundo,
atribuindo uma, usando a função *textattr*, 314
definindo, usando a função *textbackground*, 316
valores, tabela de, 316
Coreleft, função, usando, 570
Cores
primeiro plano, atribuindo com a função *textcolor*, 315
segundo plano,
atribuindo, usando a função *textattr*, 314
definindo, usando a função *textbackground*, 316
tela, controlando, 315
Corrupção no heap, definição, 1365
Cos, função, usando, 327
Co-seno hiperbólico, definição, 328
Co-seno, triângulo, definição, 327
Cosh, função, usando 328
Country, função, usando para obter informações específicas do país, 559
Cout, canal de E/S,
caractere de preenchimento, 828
compreendendo o, 806
escrevendo um caractere com 856
exibindo dígitos em ponto flutuante, 830
justificando, 829
largura, definindo com o manipulador *setw*, 827
operadores bit a bit com, 836
para combinar valores, 808
para controlar a saída com, 826
para escrever,
strings de caracteres e números, 807
valores, 807
variáveis, 807
redirecionando, 810
restaurando as definições padrão, 832
usando, 805-806
Cprintf, função, usando para saída formatada mais rápida, 297
Cputs, função, usando para saída de string na tela mais rápida, 300
Creat, função, usando para criar um arquivo, 403
CreateAcceleratorTable, usando para criar uma tabela Aceleradora, 1310
CreateBitmap, função, usando para criar mapas de bits monocromáticos, 1434
CreateCompatibleBitmap, usando para criar um dispositivo do Contexto em memória, 1421
CreateDC, função
perigos do uso, 1422
usando, para criar um dispositivo do contexto para um dispositivo, 1420
CreateDialog, macro, compreendendo, 1329
CreateDialogParam, função, compreendendo, 1330
CreateDIBitmap, 1436
CreateDirectory função, usando para criar diretórios, 1467
CreateEnhMetaFile, função, usando, 1441
CreateEvent, função, usando
para criar um evento do núcleo, 1452
para criar um evento, 1413
CreateFile, função, usando
com diferentes dispositivos, 1453
para abrir arquivo, 1452
CreateFileMapping, função, usando, 1460
CreateFont, função, usando para criar fontes personalizadas, 1423
CreateFontIndirect, função, usando,
para criar fontes personalizadas, 1423
para exibir múltiplas fontes, 1425
CreateIcon, função, usando, para criar ícones, 1445
CreateIconFromResource, usando, para criar ícones, 1446
CreateIconIndirect, função, usando para criar ícones, 1447
CreateMutex, função, usando para criar mutexes, 1410
CreateNamedPipe, função, usando para criar uma canalização nomeada, 1481
CreateProcess, função, usando para criar um processo, 1377
CreateSemaphore, usando, 1412
CreateThread, função, usando para criar um único encadeamento, 1385
CreateWindow, função, compreendendo, 1266
CreateWindowEx, função, usando, 1280

Creatnew, função, usando para criar um novo arquivo, 439
Creattemp, função, usando para criar um arquivo em um diretório específico, 438
Criação de janela, compreendendo, 1266
Cscanf, função, usando para entrada formatada mais rápida, 298
Ctime, função, usando, 622
Ctrl+Break, rotina de tratamento, criando, 775
Ctrl+Break, verificação do, usando a função *setcbrk*, 555
Ctrlbrk, função, usando, 775
CUBE, macro, criando uma, 157
Cursor, posição
determinando na tela, 307, 308
usando o controlador de dispositivo ANSI, 80

Dados binários, 1009, 1010, 1011, 1012
Dados da classe estática, 1073
Dados de arquivo, formatado, lendo, 449
Dados privados, acessando, 900
Dados,
arquivo formatado, lendo, 449
compartilhando, usando arquivos mapeados na memória, 1380
convertendo em uma construtora, 1105
privado, acessando, 900
Data do sistema
definindo a, 635
obtendo a, 633
Data juliana, determinando, 645
Data, DOS, convertendo para um formato do UNIX, 636
Dblspace, compreendendo, 353
Declaração de função-membro, compreendendo, 914
Declaração,
definição, 794
referenciando, definição, 794
hesitante, compreendendo, 793
Declarações de função *in-line*, 915
Declarações de string, 264, 265
Declarações de variáveis com o modificador *const*, 732
DefDlgProc, função, definindo uma função de processamento de Mensagem padrão, 1331
Definição de caixa de diálogo, componentes da, compreendendo, 1324
Definições de classe, colocando em um arquivo de cabeçalho, 968
Definindo a declaração,794
DefWindowProc, função, 1278
Delay, função, usando a, 624
Delete, função
para alocar matrizes, 958
para liberar as matrizes, 971
sobrecarregando a , 957
DeleteCriticalSection, função, 1406
DeleteFile, função, usando para remover arquivos de um Diretório, 1473
DeleteMenu, função, usando para excluir seleções do menu, 1309
Delline, função, usando para excluir a linha atual, 306
DELTREE, comando,
criando seu próprio, 398
usando, para remover uma árvore de diretório, 398
Depuração, definição, 9
Descritores
MENUITEM, compreendendo, 1301
POPUP, compreendendo, 1301
Descritores de arquivo, definição, 368
Deslocamento, bit a bit, executando, 94
Desreferenciando um ponteiro, definição, 512
DestroyWindow, função, usando, 1281
DEVMODE, estrutura de dados, membros, tabela de, 1420
DialogBox macro, usando para exibir uma caixa de diálogo, 1326
Difftime, função, usando, 626
Directvideo, variável global, 723
Diretivas do pré-processador, definição, 133
Diretório classificado, exibindo um, 746
Diretório(s)
árvore, removendo uma, 398
atual, mudando o, 395
classificados, exibindo, 746
criando um, 396
usando, função *CreateDirectory*, 1467
DOS,
abrindo, para funções de leitura, 431
funções de E/S, tabela de, 430
lendo, 432
trabalhando com, 430
mudando o atua., usando *SetCurrentDirectory*, 1468
recuperando o atual, usando, *GetCurrentDirectory*, 1468
removendo, 397
arquivos de um, 1473
usando a função *RemoveDirectory*, 1470
sistema Windows, lendo com *GetSystemDirectory*, 1469
Disco, operações de E/S, executando, 357
DisconnectNamedPipe, função, usando para desconectar a ponta do servidor da canalização nomeada, 1484
DispatchMessage, função, 1354
compreendendo a, 1290
usando, para processar mensagens, 1272
Display_and_change, função, usando, 226, 231
Dispositivo do contexto
compreendendo melhor, 1416
definição, 1357, 1416
função, 1421
liberando, 1430
memória, criando com a função *CreateCompatibleDC*, 1421
obtendo, para uma janela inteira, 1429
para uma janela, obtendo, 1419
privado, usando, 1417
Dispositivo, capacidades, recuperando, 1426
Dispositivos, comuns, tabelas de, 141
Dsv função, usando, 331
Divisão inteira, efetuando, 333
Divisão, executando, 333
DlgDirList, função, criando uma caixa de lista, 1332
DlgDirSelectEx, função, usando em resposta às seleções da Caixa de lista do usuário, 1333
DOS
ambiente, acrescentando elementos no, 687
comando, interno, chamando um, 730
data, convertendo, para formato do UNIX, 636
diretório,
abrindo, para funções de leitura, 431
funções de E/S, tabela de, 430
lendo, 432
diretórios, trabalhando com, 430
informações de erro, obtendo estendidas, 566
serviços de arquivo,
para acessar um arquivo, 440
usando, 410
Dosexterr, função, usando , 566
Dostounix, função, usando, 636
Double, tipo de variável, compreendendo, 35
DTA, área de transferência do disco,
acessando, 561
compreendendo, 560
controlando, 561
definição, 560
Dup, função, usando para duplicar indicativos de arquivo, 423
Dup2, função, usando para forçar a definição de um indicativo De arquivo, 424
DWORD, tipo de caractere, compreendendo, 1275
Dynamic_cast, operador, usando, 1149

E/S de arquivo, janelas, compreendendo, 1450
E/S em porta serial, controlando, 564
E/S mapeada na memória, definição, 610
E/S, porta serial, controlando, 564
E/S, redirecionamento, combinando, 653
EDIT, classe do Windows, definição, 1276
Efeitos colaterais, definição, 556
Elementos,
  acessando em uma matriz bidirecional, 470
  acrescentando ao ambiente DOS, 687
  excluindo de uma lista, 747
  inserindo em uma lista ligada dupla, 752
  matriz,
    acessando, 458
    definição, 458
    número de, determinando, 506
    passando específico, 249
    percorrendo em um laço, 459
  removendo de uma lista ligada dupla, 751
Elevador da barra de rolagem, definição, 1346
Ellipsis, operador, usando com exceções, 1135
EnableMenuItem, função, usando para controlar menu, 1307
EnableScrollBar, para habilitar ou desabilitar as barras de Rolagem, 1356
Encadeamento(s)
  apresentando, 1253
  avaliando a necessidade de, 1383
  compreendendo como o sistema operacional escalona, 1396
  compreendendo melhor, 1382
  definição, 1253, 1376
  fazendo uma pausa, 1403
  finalizando, 1394
  identificação, determinando a, 1395
  página zero, definição, 1397
  passos de criação do sistema operacional, 1387
  quando não criar, 1384
  quando usar, 1383
  reiniciando, 1403
  sincronização,
    compreendendo, 1404
    usando a função *WaitForMultipleObject*, 1409
    usando a função *WaitForSingleObject*, 1408
  tamanho da pilha, determinando, 1388
  tempo de processamento, tratando, 1390
  tipos de sincronização,
    eventos, definição, 1404
    mutex, criando, 1410
    seção crítica, criando, 1405
    semáforos, definição, 1404
    tabela dos cinco maiores, 1404
Encapsulamento, compreendendo, 887
EndDialog, compreendendo, 1334
Endereço de deslocamento e de segmento, compreendendo, 577
Endl, usando para gerar uma nova linha, 862
Ends, manipulador, compreendendo, 1039
EnhMetaFileProc, função, usando o, 1442
EnterCriticalSection, função, 1406
Entrada,
  bufferizada, compreendendo, 288
  formatada mais rápido, usando a função cscanf, 298
  lendo, a partir de uma string de caracteres, 214
  redirecionada, numerando, 657
  usuário, compreendendo, 1335
Enum, palavra-chave, usando em *C++*, 839
EnumEnhMetaFile, função, usando, 1442
EnumFontFamilies, função, usando, 1424
EnumProps, função, usando, para listas as propriedades de uma janela, 1284
EnumResourceNames, função, usando para determinar o conteúdo de um arquivo de recursos, 1316
EnumResourceType, usando para determinar os tipos de arquivo de recursos, 1317
Envi, matriz,
  como um ponteiro, 677
  usando a, 676
Environ, variável global, usando a, 685
Eof, função, usando, para testar o final de um arquivo, 405
Eof, método, usando para detectar o final do arquivo, 1016
Erro matemático, detectando, 724
Erro matemático, rotina de tratamento, criando, 349
Erros críticos,
  definição, 771
  tratando, 772
Erros de sintaxe, compreendendo, 4
Erros lógicos, compreendendo, 9
Erros,
  canal de arquivo, testando, 381
  críticos,
    compreendendo, 771
    rotina de tratamento, criando uma, 772, 773
    tratando, 772
  estendida do DOS, obtendo informações, 566
  lógicos, compreendendo, 9
  matemáticos
    detectando, 724
    rotina de tratamento, criando, 349
  sintaxe, compreendendo, 4
  sistema, detectando, 724
Erros, definição, 9
Escopo global, resolvendo, 824
Escopo,
  arquivo, definição, 277
  bloco, definição, 277
  categorias de C, compreendendo, 277
  compreendendo, 285
  definição, 225
  função, definição, 277
  gancho de mensagem, tabela de, 1295
  protótipo de função, definição, 277
  variável global, definição, 225
Espaço em branco, descartando na entrada, 966
Espaço em disco, determinando o disponível, 352
Espaço livre,
  compreendendo, 840
  liberando memória, 846
  near e *far*, 844
  testando, 843
Espaço, determinando o disponível no disco, 352
Espaços, em definições de macro, 158
Especificação de memória expandida (EMS), definição, 578
Especificações de vinculação, compreendendo, 863
Estilos de janela, registrando com a função *RegisterClass*, 1269
Estrutura da chamada, definição, 271
Estrutura da janela, definição, 1255
Estrutura de campos de bit, criando uma, 485
Estrutura(s)
  alterando, dentro de uma função, 539-540
  compreendendo, 531, 532
  contendo matrizes, 547
  declarando uma função-membro fora, 876
  definindo uma função-membro dentro, 875
  em *C++*, compreendendo, 873
  escrevendo,
    usando a função *fwrite*, 422
    usando a função *write*, 421
  inicializando uma, 544
  lendo,

usando a função *fread*, 422
usando a função *read*, 421
lista ligada, declarando uma, 739
matriz de, criando uma, 548
passando para uma função, 538
tag, definição, 533
usando uma 537
variáveis, modos de declarar, 534
versus classes, quando usar, 890
visualizando uma, 536
Eventos de auto-reinicialização, definição, 1408
Eventos, tipo de sincronização do encadeamento, definição, 1404
Exceções explícitas, pegando em um único bloco try, 1137
Exceções genéricas, pegando em um único bloco try, 1137
Exceções não-tratadas, 1393
Exceções,
lançando a partir de uma função dentro de um bloco *Try* externo, 1131
não-tratadas, tratadas, 1393
relançancando, 1139
restringindo, 1138
Execl, função, usando, 757
Executando, processo-filho, 757
ExitProcess, função, usando para desligar um processo, 1377
ExitWindowsHookEx, função, compreendendo, 1297
Exp, função, usando, 334
Exponencial trabalhando com, 334
Extensões, compreendendo, 1418
Extern, palavra-chave, usando o parâmetro, 267-68
Extratoras, usando com matrizes de canais, 1036
Extravasamento, compreendendo, 50

Fabs, função, usando, 335
Farcalloc, função, usando para alocar memória, 598
Farmalloc, função, usando para alocar memória, 598
Fatorial, processamento, tabela de, 240
Fclose, função, usando, para fechar um arquivo, 361
Fcloseall, função, para fechar todos os arquivos abertos, 384
Fdopen, função, usando para associar um indicativo de arquivo com um canal, 425
Feof, função, usando para testar o final de um arquivo, 447
Ferror, macro, usando para ler erros no canal de arquivo, 381
Fgetc, função, usando, 362
Fgets, função, usando para ler linhas de texto em um arquivo, 443
FILE, estrutura, compreendendo, 360
Filelength, função, usando para determinar o tamanho de um arquivo, 382
Fileno, função, usando, para obter um indicativo de arquivo, 385
FILES=, linha do *CONFIG.SYS*, compreendendo, 367
Filetable.c, programa, criando um, para exibir os itens da tabela de processos do arquivo, 371
Fill, função-membro, usando para gerar saída formatada, 990
Final de arquivo,
detectando, usando o método *eof*, 1016
testando, 861
Find, função, classe *linked_list*, compreendendo, 1188
FindClose, função, 1477
FindNextFile, usando para pesquisar um diretório para uma ocorrência de arquivo, 1475
FindResource, função, usando para localizar a posição do tipo de recurso, 1318
Float, tipo de variável, compreendendo, 34
Float, tipo,
valores, exibindo usando *printf*, 58
variável, compreendendo, 34
Flush, função, usando para escrever dados de um buffer para o disco, 383
Flush, usando, 819
Fmod, função, usando, 336
Fopen, função, usando para abrir um arquivo, 359
FormatMessage, função, usando para formatar as mensagens de erro, 1492
Fprintf, função, usando para efetuar saída formatada em arquivo, 375
Fprintf, função, usando para formatar as mensagens de erro, 1492
Fputc, função, usando, 362
Fputs, função, usando para escrever linhas de texto em um arquivo, 444
Fread, função, usando para ler estrutura, 422
Freopen, função, usando para reabrir um canal de arquivo, 452
Frexp, função, usando, 337
Friend, palavra-chave, usando, para especificar uma classe como uma amiga, 1043
Frwdlist, função, classe *linked_list*, compreendendo, 1187
Fscanf, função, usando para ler dados de arquivo formatados, 449
Fseek, função, usando para posicionar o ponteiro de arquivo, 450
Fstat, função, usando para obter informações do indicativo de arquivo, 450
Fstrerr, função, usando, 181
Ftell, função, usando para determinar o valor do ponteiro da posição atual, 364
Ftime, função, usando, 642
Função amiga
definição, 919
sobrecarga de operador, restrições, 953
Função construtora da cópia
definição, 937
usando, 937
Função construtora explícita, usando, 938
Função de linguagem assembly
chamando, 272
retornando um valor de, 273
Função destrutora,
compreendendo, 933, 935, 936
definição, 933
usando, 934
Função do operador-membro, criando uma, 946
Função fatorial,
definição, 241
recursivo, compreendendo, 241
Função genérica do método da Bolha, usando, 1117
Função virtual pura, definição, .094
Função,
amiga, definição, 919
C, para converter representações numéricas ASCII, 198
callback, compreendendo, 1285
classe, definindo, fora da classe, 903, 904
como membros da estrutura, 874
construtora de cópia, usando, 937
construtora explícita, usando, 938
construtora,
compreendendo, 921
conflitos de nomes, resolvendo, 926
definição, 930
sobrecarregados, encontrando o endereço de, 931
sobrecarregando, 930
usando,
com parâmetros, 922, 925
com um único parâmetro, 932
para alocar memória, 927
para inicializar membros das ocorrências, 923
valores de parâmetro padrão, 929
conversão,
criando, 1144
funções de conversão, criando, 1144
usando para aumentar a portabilidade do tipo, 1145
criando, para retornar um ponteiro, 520
definindo, dentro das classes, 904
destrutora,
compreendendo, 933, 935, 936
usando, 934
E/S no diretório do DOS, tabela de, 430

escrevendo uma, para determinar o tamanho da lista, 1200
extratora, criando, 997, 998
fila APC, 1495
genérica
compactando uma matriz, 1118
compreendendo, 1111
Método da Bolha, usando, 1117
restrições nas, 1115
sobrecarregando, 1114
usando, 1116
in-line, quando usar, 916
inserçora, criando 994
linguagem assembly,
chamando, 272
retornando um valor de, 273
manipulador, criando, 999
membro estático, usando, 913
mudando,
um membro de estrutura dentro de uma, 540
uma estrutura em uma, 539-540
operador-membro, criando um, 946
passar,
strings para, 248
uma estrutura para uma, 538
uma matriz bidimensional para, 476
uma matriz para, 461
ponteiros para,
criar, 528
usando, 529
recursiva, definição, 245
retornando referências, 852
retornando um valor de, 258
teclado, 166, 301
usando variáveis dentro, 253
virtual pura, definição, 1094
virtual
compreendendo, 1090, 1092
usando, 1096
WndProc, 1278
Função-membro estática pública, acessando, 1078
Função-membro,
compreendendo, 884
declarando, fora de uma estrutura, 876
definindo, dentro de uma estrutura, 875
estática,
compreendendo, 1077
usando, 913
estáticas públicas, acessando, 1078
fill, usando para gerar saída formatada, 990
gcount, usando, *1013*
ios flags, usando para examinar as definições atuais de formato, 987
ios precision, usando para formatar a saída, 989
ios, usando para manipular matrizes de canais, 1031
passando parâmetros para um 877
pcount, usando, com matrizes de saída, 1030
rdstate, usando para testar o status de E/S atual, 1024
versus manipuladores, 993
Funções construtoras
com parâmetros, 922, 925
com um único parâmetro, 932
conflito de nome, resolvendo, 926
definição, 930
para inicializar membros de ocorrências, 923
sobrecarregada, encontrando o endereço, 931
sobrecarregando, 921
usando
para alocar memória, 927
valores de parâmetro padrão, 929
Funções de atribuição, classe Strings, escrevendo as, 1168
Funções de callback, compreendendo, 1262, 1285
Funções de classe, definindo, fora da classe, 903, 904
Funções de conversão,
criando, 1144
usando para melhorar a portabilidade do tipo, 1145
versus operadores sobrecarregados, 1146
Funções de interface, definição, 902
Funções do heap, usando para gerenciar memória, 1369
Funções extratoras, criando, 997, 998
Funções *in-line*, quando usar, 916
Funções inserçoras, criando, 994
Funções intrínsecas,
desabilitando e habilitando, 779
in-line, 778
Funções manipuladoras, criando, 999
Funções virtuais,
compreendendo, 1090, 1092
usando, 1096
Funções-membro estáticas,
compreendendo, 1077
usando, 913
Fwrite, função, usando para escrever estruturas, 422

Gabarito(s)
colocando em uso, 1110
com múltiplos tipos genéricos, 1113
compreendendo, 1109
onde colocar, 1119
que suportam múltiplos tipos, 1112
usando para eliminar classes duplicadas, 1120
Ganchos de mensagem, definição, *1295*
Gcount, função-membro, usando, 1013
Geninterrupt, função, usando a, 769
Gerador de números aleatórios, alimentando o, 347
Gerenciador do heap local, introdução ao, 1358
Get, funções, sobrecarregadas, 1014
Get, método, usando para ler dados binários, 1009
GetAsyncKeyState, usando para determinar o status das teclas, 1345
Getcbrk, função, usando para obter o estado da verificação Ctrl+Break, 555
Getchar, macro,
combinando com a macro *putchar*, 290
compreendendo o 656
usando, para ler um caractere do teclado, 286
Getche, função, usando para ler um caractere do teclado, 292
GetCurrentDirectory, usando para recuperar os diretórios atuais, 1468
GetCurrentProcess, compreendendo, 1389
GetCurrentProcessID, usando para determinar um identificador de processo, 1395
GetCurrentThread, função, compreendendo, 1389
GetCurrentThreadID, função, usando para determinar o Identificador do encadeamento, 1395
Getdata, função, usando para controlar o área de transferência do disco, 561
Getdate, função, usando, 633
GetDC, função, usando para obter o dispositivo do contexto para uma janela, 1419
GetDeviceCaps, função, usando, para obter o dispositivo do contexto para uma janela, 1419
GetDeviceCaps, usando, para recuperar informações específicas do dispositivo, 1426
GetDoubleClickTime, função, usando a, 1343
Getend, função, classe *linked_list*, compreendendo a, 1186
Getenv, função
usando, 683
para abrir arquivos no diretório *TEMP*, 392
GetExitCodeProcess, função, usando para recuperar valores de saída do processo, 1378
Getfat, função, usando para ler a tabela de alocação de arquivos, 354

Getfatd, função, usando para ler as informações da tabela de alocação de arquivos, 354
GetFileAttributes, função, usando para obter o tamanho dos arquivos, 1464
GetFileAttributes, usando para obter os atributos dos Arquivos, 1464
GetFileSize, função, usando para obter o tamanho do arquivo, 1465
GetFileTime, função, usando para obter a hora do arquivo, 1466
GetLastError, função, compreendendo, 1491
Getline, método, usando, 1015
GetMessage, função, 1270, 1354
Getnext, função, classe *list_object*, compreendendo, 1180
Getpass, função, usando a, 649
Getprevious, função, classe *list_objects*, compreendendo a, 1180
GetPriorityClass, função, usando para retornar a classe de prioridade de um processo, 1399, 1402
GetProcessTimes, usando para verificar o tempo de execução de múltiplos encadeamentos, 1391
GetqueueStatus, função, usando, para determinar o conteúdo da fila de mensagem, 1392
Gets, função, usando,
para ler uma string de caracteres do teclado, 301
para ler strings de caracteres do teclado, 166
GetScrollInfo, função,
usando, para obter as definições da barra de rolagem atual, 1351
usando, para ler a posição e o intervalo da posição da barra de rolagem, 1349
Getstart, função, classe *linked_list*, compreendendo a, 1186
GetSystemDirectory, função, usando para ler o diretório do sistema Windows, 1469
GetSystemMetrics, função,
compreendendo o uso para, 1428
usando, para analisar uma janela, 1427
GetSystemPaletteEntries, usando, 1438
GetTempFileName, usando para nomear um arquivo temporário, 1480
GetTempPath, função, usando para ler um caminho temporário, 1479
Gettextinfo, função, usando para determinar as definições do modo texto, 312
GetThreadPriority, função, usando para ler o valor de prioridade de um encadeamento, 1385
GetThreadTimes, função, usando para ler o valor de prioridade de um encadeamento, 1390
Gettime, função, usando, 632
GetUpdateRect, função, 1354
Getw, função, usando para ler uma palavra de um arquivo, 415
GetWindowDC, função, usando para obter o dispositivo do contexto para uma janela inteira, 1429
GetWindowsDirectory, função, usando para obter o diretório do Windows, 1469
GetWinMetaFileBits, usando a, 1443
GlobalAlloc, função, 1358-60
compreendendo o uso de, 1362
usando para alocar memória a partir do heap global,1362
GlobalDiscard, função, usando para liberar a memória anteriormente alocada, 1364
GlobalFree, função, usando para liberar a memória, 1365
GlobalHandle, função, usando para converter um ponteiro em um indicativo de memória, 1366
GlobalLock, função, usando para converter indicativos de memória em ponteiros, 1366
GlobalReAlloc, função,
compreendendo o uso de, 1363
usando para realocar a memória, 1363
GlobalMemoryStatus, função, usando para verificar o status da memória do computador, 1367
Gmtime, função, usando a, 631
Goto não-local,
definição, 728
efetuando uma, 728
Goto, comando, usando, para ir a uma posição específica, 128
Gotoxy, função, usando para posicionar o cursor na tela, 307
Gtpid, função, usando, 729

Halloc, função, usando, 600
Hdc, parâmetro,
CreateCompatibleDC, 1421
GetDeviceCaps, função, 1426
Heap privado, definição, 1361
Heap,
compreendendo, 597
criando, dentro de um processo, 1368
HEAP_NO_SEARIALIZE, sinalizador, 1368-70
HeapAlloc, função, 1361
Heapcheck, função, usando, 604
Heapchecknode, função, usando, 606
HeapCreate, função, 1361
usando, para criar um heap dentro de um processo, 1368
HeapDestroy, função, 1361
Heaps, revisitando, 1361
Heapwalk, função, usando, 607
Herança da classe-base protegida, compreendendo, 1055
Herança,
múltupla,
classe-base protegida, 1055
classe-base protegida, compreendendo, 1055
compreendendo, 1056
compreendendo, 889
definição, 1056-1058
ilustração de 1057
revisando, 1054
Herança, em C++, 1048
Hfree, função, usando, 600
Hierarquia de herança, criando um exemplo de, 1071
HIGH_PRIORITY_CLASS, classe de prioridade do Win32, 1398
Hora de gravação ou modificação, 411, 413, 414
Hora do sistema
definindo a, 634
obtendo a, 632
Horas, comparando, 626
Huge, modelo de memória,
compreendendo, 619
definição, 466

Ícones,
carregando em um programa, usando *LoadIcon*, 1448
compreendendo, 1444
criando,
a partir de um recurso, 1446
usando,
a função *CreateIcon*, 1445
a função *CreateIconFromResource*, 1446
a função *CreateIconIndirect*, 1447
definição, 1444
predefinidos do Windows, tabela de, 14448
ICONINFO, estrutura, 1447
ID do disco, compreendendo, 355
Identificador do processo,
determinando, 1395
obtendo, 729
Identificador,
definição, 278
visibilidade, definição, 279
IDLE_PRIORITY_CLASS, classe de prioridade do Win32, 1398
If, comando, usando para testar uma condição, 99
Ignore, função, usando, 1017
Impressora, acessando usando a BIOS, 554
Inclusões circulares, evitando, 802

Indexação segura de matriz, 959
Indicativo da janela,
compreendendo, 1259
tipos, definição, *1260*
Indicativos de arquivo
associando com um canal, 425
compreendendo, 369
definição, 1454
definição, 369
definindo, forçando, 424
duplicando um 423
obtendo, usando a função *filena*, 385
stdaux, usando, 666
stdin, introdução a, 652
stdin, usando, 654
stdout, introdução a, 651
stdout, usando, 654
stdprn, usando, 664
usando, 1454
versus ponteiros de canais, compreendendo o relacionamento, 374
Indução, laço, compreendendo, 786
Informações de indicativo de arquivo, obtendo, usando a Função *fstat*, 451
Inicializada *versus* matrizes não-inicializadas, criando, 974
Inicializadores, compreendendo, 791
InitializeCriticalSection, função, *1406*
In-line, código de classe, 977
Inline, palavra-chave,
C++, usando, 853
usando, com funções membro da classe, 969
Inport, função, usando, 611
Inserçoras, usando com matrizes de string, 1035
Insline, função, usando, para inserir uma linha em branco na Tela, 309
Int, tipo,
valores, exibindo usando *printf*, 54
variável, compreendendo, 32
Interface de dispositivo gráfico (*GDI*),
compreendendo a, 1414
razões para usar, 1415
Interface de documento no estilo do Explorer, definição, 1256
Interface do programa, definição, 902
Interrupções
alteradas, restaurando, 774
compreendendo, 761
definição, 553
desabilitando, 766
gerando um, 769
habilitando, 766
PC, tabela de, 762
software, compreendendo, 553
Interrupções de software, compreendendo, 553
Interrupções do PC, tabela, 762
Interrupt, palavra-chave, usando, 763
Invariante, definição, 782
Inverso, operador bit a bit (~), compreendendo, 89
Ios flags, função-membro, usando para examinar as definições atuais de formato, 987
Ios precision, função-membro para formatar a saída, 989
Ios, classe, *980*
Ios, funções-membro, usando para manipular matrizes de canal, 1031
Ios, membros, usando para formatar a entrada e saída, 983
Ios, sinalizadores de formato, 983
Iostream.h, arquivo de cabeçalho, compreendendo, 820
Isalnum, macro, compreendendo, 199
Isalpha, macro, usando para determinar se um caractere é do Alfabeto, 200
Isascii, macro do arquivo de cabeçalho *ctype.h*, 201
Iscntrl, macro do arquivo de cabeçalho *ctype.h*, 202
Isdigit, macro do arquivo de cabeçalho *ctype.h*, 203
Isgraph, macro do arquivo de cabeçalho *ctype.h*, 204
Islower, macro do arquivo de cabeçalho *ctype.h*, 205
Isprint, macro do arquivo de cabeçalho *ctype.h*, 206
Ispunct, macro do arquivo de cabeçalho *ctype.h*, 207
Isspace, macro do arquivo de cabeçalho *ctype.h*, 208
Istrstream,
sobrecarregado, usando, 1029
usando, para escrever uma string de caracteres, 1027
Istype, macro do arquivo de cabeçalho *ctype.h*, 722
Isupper, macro do arquivo de cabeçalho *ctype.h*, 205
Isxdigit, macro do arquivo de cabeçalho *ctype.h*,, 209
Itens de propriedade, definição, 1283

Janela,
componentes de, compreendendo, 1255
dispositivo do contexto, obtendo, 1419
filha, compreendendo, 1256
mãe, compreendendo, 1256
texto, definindo uma, 320
Janela-filha, compreendendo, 1256
Janela-mãe, compreendendo, 1256
Janelas de estilo estendido,
criando, 1280
definição, 12800
Janelas,
classes de prioridade, compreendendo, 1398
criando, usando a função *CreateWindow*, 1267
destruindo, 1281
E/S em arquivo, compreendendo, 1450
estilo estendido, criando, 1280
exibindo, usando a função *ShowWindow*, 1268

Laço, terminando, usando o comando *break*, 127
Large, modelo de memória, compreendendo, 618
Ldexp, função, usando, 338
LeaveCriticalSection, função, 1406
Legibilidade, e usando operadores para, 1108
Lfind, função, usando para pesquisar uma matriz, 502
Vinculação, compreendendo, 792
Linkeditor,
arquivos de resposta, usando, 701
capacidades, vendo, 699
compreendendo, 698
Linkeditor, programa,
compreendendo, 12
definição, 11
Linguagem de programação, definição, 1
Linguagem, mistura, exemplo de, 238
Linhas redirecionadas, exibindo um contador de, 660
Linhas,
exibindo as primeiras, da entrada redirecionada, 668
redirecionada, exibindo um contador de, 660
Linhas, matriz, compreendendo, 469
Linked_list, classe
bkwdlist, função, compreendendo, 1188
compreendendo, 1183
find, função, compreendendo, 1188
frwdlist, função, compreendendo, 1187
genérica, compreendendo, 1193
getend, função, compreendendo, 1186
getstart, função, compreendendo, 1186
remove, função, compreendendo, 1185
store, função, compreendendo, 1184
Linked_list, objeto,
exibindo,
em ordem crescente, 1187
em ordem inversa, 1188
pesquisando, 1189

Linked_list, programa, criando um simples, 1190
List_object, classe,
criando, 1179-1182
getnext, função, compreendendo, 1180
getprevious, função, compreendendo, 1180
herdando a, 1182
tornando genérica, 1192
Lista de equipamentos da BIOS, obtendo a, 563
Lista ligada dupla,
criando, usando uma classe de C++ para, 1178-1190
criando uma, 749
Lista ligada simples, definição, 748
Lista ligada,
criando uma, 740
definição, 738
duplamente,
criando uma, 749
inserindo um elemento na, 752
removendo um elemento da, 751
usando uma, 748
estrutura, declarando uma, 739
exemplo, 741
percorrendo, 742
Lista, removendo um elemento da, 747
LISTBOX, classe, Windows, definição, 1276
LoadIcon, função, usando para carregar ícones em um programa, 1448
LoadImagem, função, usando, 1449
LoadMenu, função, 1302
compreendendo, 1305
LoadString, função, usando para carregar tabelas de strings em programas, 1315
Local_values, função, usando, 218
LocalAlloc, função, 1358-60
Localtime, função, usando, 630
Lock, função, usando para bloquear o conteúdo de um arquivo, 428
Locking, função, usando para bloquear o conteúdo de um arquivo, 429
Log, função, usando, 339
Log10, função, usando, 340
Logaritmo natural, calculando, 339
Long int, valores do tipo, exibindo com printf, 57
LPCTSTR, tipo de caractere, compreendendo, 1274
Lsearch, função, usando para pesquisar valores em uma matriz, 503
Lseek, função, usando para posicionar o ponteiro de arquivo, 408
Lvalue modificável e não-modificável, definição, 795
Lvalue,
compreendendo, 795
definição, 795

Macros
alfabeto, 200
anulando a definição, 243
chamando e alterando simultaneamente, 719
comparando, 144
criando, 154
definições, espaços nas, 158
nomeando, 134
Maiúsculas, convertendo uma string, usando a função *strupr*, 175
MAKE, arquivo
colocando múltiplas dependências, 706
comentando, 705
criando um simples, 703
incluindo um segundo, 713
terminando, com um erro, 715
MAKE, definição das regras, 708
MAKE, executando processamento condicional com, 711
MAKE, macro,
definição, 709
predefinida, 710
testando, 712
usando, 709
MAKE, modificadores de macro, usando, 714
MAKE, usando o utilitário, 702
Makestr, função-membro, 1173, 1174
Malloc, função, usando, 593
para alocar memória, 593
para liberar memória, 595
Manipulador parametrizado,
criando um, 1124
usando, 1001
Manipulador sem parâmetros, criando, 1000
Manipuladores stream, 992
Manipuladores,
canal de saída, definição, 1100
compreendendo, 991
definição, 991
ends, compreendendo, 1039
parametrizado,
criando, 1124
usando, 1001
sem parâmetros, criando, 1000
Setios flags, usando, 831
setprecision, usando, 830
setw, usando para definir a largura de cout, 827
usando,
com matrizes de canais, 1034
para formatar a E/S, 992
versus funções-membro, 993
Mapa de página, processo, 1358
Mapa do vínculo,
definição, 698
usando, 700
Mapas de bits
coloridos, criando, usando *CreateCompatibleBitmap*
criando, 1434
dependência do dispositivo, definição, 1432
exibindo, 1435
independência do dispositivo
criando, 1436
compreendendo, 1433
monocromático, criando com a função *CreateBitmap*, 1434
operações de varredura comuns efetuadas com, 1435
Mapas de bits dependentes do dispositivo, definição, 1432
Mapas de bits independentes do dispositivo,
criando, 1436
definição, 1432
compreendendo, 1433
Mapeamento de arquivo, definição, 1459
MapViewOfFile, função, usando, 1461
Matriz de classe,
destruindo uma, 971
inicializada, criando, 972
com construtoras de múltiplos argumentos, 973
inicializando uma, 970
trabalhando com, 975
Matriz *stream*,
formatando, compreendendo, 1038
manipulando, usando a função-membro *ios*, 1031
Matrizes dinâmicas, usando, com canais de E/S, 1037
Matrizes não-inicializadas *versus* inicializadas, 974
Matrizes,
bidimensionais,
acessando elementos em uma, 470
passando para uma função, 476
percorrendo em um laço, 473
canal, manipulando, usando as funções-membro *ios*, 1031
classificando uma, 491, 505

colunas, compreendendo, 469
compactar, usando funções genericas para, 1118
comparando, 591
de estruturas, criando uma, 548
de ponteiros, compreendendo, 521
de variáveis de classe, criando uma, 943
declarando uma, 454
definição, 453
definindo, usando constantes para, 460
dinâmica, usando com canais de E/S, 1037
env, usando, 676
como um ponteiro, 677
estruturas que contêm, 547
inicializada versus não-inicializada, criando, 974
inicializadas, criando, 972
com construtoras de múltiplos argumentos, 973
inicializando uma, 970
linhas, compreendendo, 469
multidimensional,
determinando a memória consumida, 472
compreendendo, 468
inicializando, 475
pesquisando um valor especifico na, 488
requisitos de armazenagem, compreendendo, 456
trabalhando com, 975
tridimensional, percorrendo uma, 474
uso de memória, compreendendo, 976
versus memória dinâmica, compreendendo, 467
visualizando uma, 455
Max, macro do arquivo de cabeçalho *stdlib.h*, 341
MAX, macro, criando uma, 156
Maximizar, botão, definição, 1255
Medium, modelo de memória, compreendendo, 616
Membro da estrutura,
alterando dentro de uma função, 540
compreendendo, 535
Membro protegido,
usando, em classes derivadas, 1072
usando, para limitar o acesso aos membros da classe, 1072
Membros da classe estática, compreendendo, 911
Membros da classe protegida, usando, 1052, 1053
Membros de dados estáticos,
acessando diretamente, 1075
inicializando, 1074
usando, 912, 1073
Membros de dados não-estáticos, usando, 1073
Membros de dados privados estáticos, 1076
Membros de dados, estáticos, usando, 912
Membros,
classe estática, compreendendo, 911
classe protegida, usando, 1052, 1053
dados estáticos,
acessando diretamente, 1075
inicializando, 1074
usando, 1073
dados privados estáticos, 1076
conflitos no nome do parâmetro, resolvendo, 942
estrutura, alterando dentro de uma função, 540
protegidos, usando em classes derivadas para limitar o acesso aos membros da classe, 1072
Memcmp, função, usando, 591
Memcpy, função, usando, 589, 590
Memicmp, função, usando, 591,
Memmove, função, usando, 589
Memória convencional,
acessando, 575
BIOS, determinando, 567
compreendendo, 574
layout, compreendendo, 574
Memória da pilha, alocando, 599
Memória dinâmica,
alocando, 593
versus matrizes, compreendendo, 467
Memória do computador, verificando, 1367
Memória estendida,
acessando, 582
compreendendo, 580
Memória expandida,
compreendendo, 578
usando, 579
Memória global, compreendendo, 1359
Memória local, compreendendo, 1359
Memória privada, acessando, 900
Memória virtual,
compreendendo, 1360, 1373
estados da, 1371
liberando, 1374
sinalizadores de segurança para alocações de páginas, 1371
Memória,
classes de matrizes, compreendendo, 976
computador verificando a, 1367
dinâmica, alocando, 593
global, compreendendo, 1359
liberando,
não necessário, 595
um bloco alocado, 1364
local, compreendendo, 1359
não-usada, determinando, 570
pilha, alocando, 599
privada, definição, 1359
realocando, 1363
salvando, com uniões, 482
tipos, compreendendo, 572
virtual,
compreendendo, 1360, 1373
liberando, 1374
Memset, função, usando, 588
Mensagem de erro
exibindo, atual, usando a diretiva *#error*, 138
formatando, usando a função *FormatMessage*, 1492
predefinida, exibindo, 725
Mensagens da barra de rolagem, compreendendo, 1355
Mensagens do mouse do Windows, tabela de, 1336
Mensagens do mouse, janelas, tabela de, 1336
Mensagens do teclado do Windows, tabela de, 1339
Mensagens,
compreendendo, 1254
erro predefinido, exibindo, 725
geradas pelo Windows após a chamada à *CreateWindow*, 1278
o fluxo das, compreendendo, 1289
Menu,
acrescentando à janela de um aplicativo, 1302
alterando, dentro de um aplicativo, 1303
criando, dentro de um arquivo de recurso, 1300
estrutura, compreendendo, 1299
tipos, compreendendo, 1298
MENUITEM, descritores, compreendendo, 1301
MessageBeep, função, compreendendo, 1287
MessageBox, função, compreendendo, 1286
Meta-arquivo
compreendendo, 1440
criando e exibindo, 1441
expandido, 1440, 1442
Meta-arquivo expandido,
definição, 1440
enumerando, 1442
Método da Bolha, algoritmo

compreendendo, 492
usando para classificar uma matriz, 493
Métodos, 881, 904
Métrica do sistema, definições da, tabela, 1427
MIN, macro, criando uma, 156
Minimizar, botão, definição, 1255
Minúscula, convertendo uma string para, usando a função *Strlwr*, 175
Mkdir, função, usando para criar um diretório, 396
Mktemp, função, usando para criar um nome de arquivo único, 420
Mktime, função, usando, 644
Modelo de memória,
Compact, compreendendo, 617
compreendendo, 613
determinando o atual, 620
Huge, compreendendo, 619
Large, compreendendo, 618
Medium compreendendo, 616
Small, compreendendo, 615
Tiny, compreendendo, 614
Windows 95, compreendendo, 1358
Windows NT, compreendendo, 1358
Modelo de mensagens do Windows, 1254
Modelo lógico processo-encadeamento, 1382
Modf, função, usando, 342
Modificadores de macro, MAKE, usando, 714
Modificadores de tipo,
compreendendo, 37
definição, 37
long e *unsigned*, combinando, 40
long, compreendendo, 39
registrador, compreendendo, 42
short, compreendendo, 43
signed, compreendendo, 43
unsigned, compreendendo, 38
ModifyMenu, função, 1303, 1306
Modo de acesso de um arquivo, definindo, 379
Modo protegido, compreendendo, 581
Modo real, compreendendo, 581
Modos,
protegido, compreendendo, 581
real, compreendendo, 581
Módulo aritmético, compreendendo, 82
MORE, comando,
criando um, 659
temporizado, criando um, 662
MoveFile, função, usando para mover e renomear arquivos, 1472
MoveText, função, usando para mover texto na tela, 319
MSG, estrutura, componentes, compreendendo, 1290
Múltipla herança, definição, 1056-58
Múltiplas condições, testando, 129
Múltiplas matrizes, alocando, usando o operador new, 842
Mutable, palavra-chave,
compreendendo a, 1158
usando, dentro de uma classe, 1.59
Mutex, tipo de sincronização de encadeamento, definição, 1404
Mutexes, tipo de sincronização de encadeamento, criando, 1410

Name space, definição, 278
Namespace, palavra-chave, usando, 1153
New, função,
para alocar matrizes, 958
sobrecarregando a, 956
New, operador,
alterando o tratamento padrão de, 870
usando,
para alocar memória, 841
para alocar múltiplas matrizes, 842
Níveis de prioridade, apresentando, 1397
Nodo->anterior->proxima, compreendendo, 750
Nome de arquivo,
exibindo, usando a função *show_directory*, 435
temporário,
criando,
usando a função *tempnam*, 387
usando a função *tmpnam*, 386
único, criando um, usando a função *mktemp*, 420
NORMAL_PRIORITY_CLASS, classe de prioridade do Win32, 1398
Nova linha, gerando, usando *endl*, 862
Numérico de ASCII funções, 188
Número aleatório,
gerador de, alimentando, 347
gerando um, 345
Número aleatório, gerando, 345
Número da linha atual, mudando com a diretiva *#line*, 137
Número de versão, sistema operacional, determinando, 726
Numeros da linha de comando, trabalhando com, 679
Números, linha de comando, trabalhando com, 679

OBJ arquivos, compreendendo, 691
Objeto de dispositivo do núcleo, processamento assíncrono com, 1488
Objeto de evento do núcleo, processamento síncrono com, 1493
Objeto de mapeamento de arquivo, definição, 1459
Objetos ocultos, 849
Objetos,
arquivos, definição, 11
classe, 881
código, reutilizando, 691
compreendendo, 881, 885
definição, 884
ocorrências, compreendendo, 905, 906
ocultos, 849
Strings,
convertendo, para uma matriz de caracteres, 1173
demonstrando o uso, 1175
determinando o tamanho de, 1172
usando, como uma matriz de caracteres, 1174
usando,
com a função *store*, 1199
vantagens de, 883
Ocultando informações, compreendendo, 897
Open, função, usando para abrir um arquivo, 402
Open, membro, usando para abrir um canal de arquivo, 1003
Opendir, função, usando para abrir um diretório do DOS, 431
OpenFileMapping, função, usando, 1462
OpenMutex, função, usando para obter um indicativo para um mutex criado anteriormente, 1411
Operação de cópia binária, executando, 1008
Operação, aplicando uma, ao valor de uma variável, 91
Operações bit a bit
definição, 87-90
usando com, canal de E/S *cout*, 836
Operações de abertura de arquivo, controlando, 417
Operações de E/S,
disco, efetuando, 357
verificando o status das, 1006
Operações de varredura,
comuns, efetuada com mapas de bits, 1435
PatBlt, função, tabela de, 1437
Operações matemáticas, básicas em C, compreendendo, 81
Operações,
canal de arquivo, combinando, 1007
controlando abertura de arquivo, 417
Operador bit a bit de inversão (~), compreendendo, 89
Operador condicional de C, (?), compreendendo, 92
Operador de atribuição (+), sobrecarregando, 1169
Operador de canalização, compreendendo o, 655

Operador de chamada de função (), sobrecarregando, 960
Operador de comparação (=), sobrecarregando, 1197
Operador de decremento,
C, compreendendo, 86
sobrecarregando, usando uma função amiga, 950, 954
Operador de definição de escopo, compreendendo, 894
Operador de deslocamento para a direita (>>), 94
Operador de deslocamento para a esquerda (<<), 94
Operador de extração, sobrecarregando o, 995.
Operador de incremento,
C, compreendendo, 85
sobrecarregando, 949
usando um função amiga, 954
Operador de indireção (*), usando, para desrefetenciar o valor de um ponteiro, 512
Operador de inserção, sobrecarregando para exibir strings em maiúsculas, 996
Operador de (+), sobrecarregando, 947
Operador de resolução global (::),
revisando o, 908
usando, 824
Operador *E* (&) bit a bit, compreendendo, 89
Operador *OU* bit a bit (|), compreendendo, 87
Operador *OU* Exclusivo (^), compreendendo, 89
Operador vírgula
compreendendo, 962
sobrecarregando, 962
Operador,
atribuição, múltipla, 46
condicional, compreendendo 96
const_cast, usando o, 1148
conversão, C++, compreendendo a, 1147
decremento, sobrecarregando, 950
usando uma função amiga, 954
deslocamento, 94
extração, sobrecarregando, 995
quando usar para maior legibilidade, 1108
resolução de escopo, compreendendo, 894
static_cast, usando o, 1151
typeid,
usando, para a identificação de tipo, 1156
valores de retorno, 1157
Operadores condicionais, compreendendo, 96
Operadores de atribuição, múltiplos, 46
Operadores de conversão de C++, compreendendo, 1147
Operadores relacionais, sobrecarregando, 1171
Origem, definição, 1418
Ostrstream,
compreendendo, 1028
usando, para criar uma matriz dinâmica, 1037
OU Exclusivo, operador bit a bit (^), compreendendo, 89
OU, operador bit a bit (|), compreendendo, 87
Outtext, função, usando para exibir saída colorida na tela, 303
OVERLAPPED, estrutura, 1487
Overlay, 760

PAGE_GUARD, sinalizador, compreendendo, 1371-72
Páginas de guarda, compreendendo, 1372
Palavra-chave
asm, C++, usando, 854
auto, compreendendo, 276
BITMAP usando para acrescentar mapas de bits a um arquivo de recursos, 1433
C++,
lista de, 31
tabela de novos, 822
C, compreendendo, 31
cdecl, compreendendo, 239
const,
compreendendo o uso, 250
definição, 30
usando em C++, 838
enum, usando em C++, 839
explicit, usando, 938
extern, usando, 267, 268
friend, usando para especificar uma classe como amiga, 1043
in-line,
C++, usando, 853
usando, com funções-membro da classe, 969
interrupt, usando, 763
mutable,
compreendendo, 1158
usando dentro de uma classe, 1159
namespace, usando, 1153
pascal,
compreendendo, 237
usando, 236
para criar uma classe derivada privada, 1059
public, usando, para criar uma classe derivada pública, 1059
registrador de segmento, 797
static, usando, 269
ao declarar membros da classe, 1073
para declarar variáveis, 234
struct, em C++, 873
virtual, usando, 1069
void, usando, 275, 678
volatile, compreendendo, 270
Palavras-chave de C
compreendendo, 31
lista de, 31
Parágrafos, definição, 577
Parâmetro formal, compreendendo, 262
Parâmetros da função, usando ponteiros com, 514
Parâmetros,
declarando, 257
definição, 154, 252, 255
e conflitos de nome do membro, resolvendo, 942
formal, compreendendo, 262
função, usando ponteiros com, 514
introdução aos, 255
múltiplos, usando, 256
passando,
para construtoras de classe-base, 1065
para uma função-membro, 877
usando três técnicas diferentes, 850
reais, compreendendo, 262
valores,
alterando, 231
fornecendo padrões, 825
Parênteses, usando, compreendendo, 159
Pascal, modificadores, compreendendo, 801
Pascal, palavra-chave,
compreendendo, 237
usando, 236
Password, função, criando uma, 650
PatBlt, função,
operações de varredura, tabela de, 1437
usando, para desenhar retângulos em um dispositivo do contexto, 1437
Pausa, intervalo, especificando, usando a função *sleep*, 557
Pcount, função-membro, usando com matrizes de saída, 1030
Peek, função, usando, 608, 1018
Peekb, função, usando, 608
PeekMessage, função, 1354, 1291
Pequisa seqüencial, 488
Pesquisa binária,
compreendendo uma, 489
usando uma, 490

Pesquisa binária, 489, 490
Pilha,
como as funções usam a, 219
compreendendo a, 584
determinando a atual do programa, 586
PlayEnhMetaFile, função, usando a, 1441
Pointer, operador, sobrecarregando, 961
Poke, função, usando, 609
Pokeb, função, usando, 609
Polimorfismo,
compreendendo 888
definição, 1093
implementação, 1093
Ponteiro da função, *_new_handler*, usando, 870
Ponteiro da posição,
compreendendo, 363
valor, determinando o atual 364
Ponteiro de arquivo, posicionando,
usando a função *fseek*, 450
usando a função *lseek*, 408
Ponteiro *far*
compreendendo, 798
criando um, 568
definição, 568
Ponteiro *void*, compreendendo, 527
Ponteiro(s)
alocando, para uma classe, 964
canal de arquivo, controlando o, 1021
decrementando um, 516
definição, 63, 507
descartando, para uma classe, 965
desreferenciando, 512
endereços, exibindo usando a função *printf*, 63
far, criando um, 568
incrementando um, 516
matriz de, compreendendo, 521
normalizado, compreendendo, 799
para classes, 1087
para diferentes classes, 1088
para funções, criando, 528
para funções, usando, 529
posição, determinando, usando a função *tell*, 436
posição, encontrando a atual, 1020
retornando um, criando uma função para, 520
this, compreendendo, 1084, 1085
usando,
para declarar uma constante string, 526
para percorrer uma string em um laço, 528
com parâmetros da função, 514
valores, usando, 513
variável, declarando uma, 230, 511
versus declarações de string, compreendendo, 265
void, compreendendo, 527
Ponteiro, ->membro, usando, 541
Ponteiros normalizados, compreendendo, 799
Ponto-e-vírgula,
compreendendo 22
em macros, 155
Ponto flutuante arredondando para cima, 326
POPUP descritores, compreendendo, 1301
Porta, valores, acessando, 611
Portabilidade, 727
Portas de finalização de E/S,
introdução às, 1499
usando, 1500
Portas do PC, compreendendo, 610
PostMessage, função, compreendendo, 1292
Pow, função, usando, 343
Pow10, função, usando, 344
Pragma do compilador, compreendendo, 145
Precedência de operador,
compreendendo 83
forçando, 84
Precedência dos operadores, 83, 84
Precisão, compreendendo, 51
Preenchimento com zeros, definição, 66
Prefixo do Segmento do Programa (PSP), definição, 675, 731
Primeiro comando, escrevendo um simples, 859, 860
Printf, função,
caracteres exibidos, determinando o número de 75
compreendendo, 53
direcionando para exibir o sinal de um valor, 64
especificadores de formato, combinando, 71
saída, justificando à esquerda, 70
usando
para exibir endereços de ponteiro, 63
para exibir strings de caracteres, 62
para exibir valores *char*, 59
para exibir valores em ponto flutuante, 60
para exibir valores *float*, 58
para exibir valores *int*, 54
para exibir valores *long int*, 57
para exibir valores *unsigned int*, 56
para formatar valores em ponto flutuante, 68
para formatar valores inteiros, 65
valor de retorno, usando, 76
Prioridade da classe
alterando, 1399
janelas, compreendendo, 1398
níveis, 1398
Private, palavra-chave, usando para criar uma classe derivada privada, 1059
Private rótulo, usando o, 898
Processador de evento, usando um simples, 1413
Processamento assíncrono,
compreendendo melhor, 1485
quatro técnicas de, 1486
Processamento condicional
definição, 96
efetuando com o MAKE, 711
Processamento iterativo, compreendendo, 97
Processamento,
condicional, efetuando com o MAKE, 711
iterativo, compreendendo, 97
Processo,
classe de prioridade,
alterando com a função *SetPriorityClass*, 1401
obtendo a atual, 1402
criando, 1377
definição, 1376
Processo-filho destacado, definição, 1381
Processo-filho,
compreendendo, 753
definição, 753, 1379
destacado, definição, 1381
gerando um, 754, 1379
Programa C
compilando, 3
estrutura, 5
Programa de computador, definição, 1
Programa de filtro, escrevendo um simples, 857
Programa principal, definição, 5
Programa,
atual, abortando, 688
comandos,
acrescentando, 6
atribuindo, 5
excluindo usando comentários, 20

comentando, 16
compilando um C, 3
computador, definição, 1
estrutura de, 5
janela versus sem janelas, 1252
melhorando a legibilidade, 17
processo de desenvolvimento, compreendendo, 10
quebrando em objetos, 884
Windows,
componentes de, compreendendo, 1273
genérico, criando um, 1257
Programação orientada a objetos, compreendendo, 882
Protected, rótulo, usando o, 899
Protótipo de função,
compreendendo, 260
em C++, definição, 821
Public, palavra-chave, usando para criar uma classe derivada pública, 1059
Public, rótulo, compreendendo, 896
Putchar, macro,
combinando com a macro *getchar*, 290
compreendendo a, 656
usando, para escrever um caractere na tela, 287
Put, método, usando para escrever dados binários, 1010
Putback, função, usando, 1019
Putch, função, usando para executar saída rápida na tela, 295
Putenv, função, usando, 686
Puts, função, usando para escrever uma string de caracteres, 299
Puttext, função, usando para colocar texto na tela 311
Putw, função, usando para escrever uma palavra em um arquivo, 415
Qsort, função, usando para classificar matrizes, 505
QuickSort,
compreendendo, 498
usando, para classificar uma matriz, 499

R_strstr, função, usando para encontrar a ocorrência mais à direita de uma substring, 194
Raiz quadrada, calculando, 348
Rand, função, usando, 345
Random, função, usando a, 345
Randomize, função, usando, 347
Rdstate, função-membro, usando para testar o status de E/S Atual, 1024
Read, função, usando,
para ler dados binários, 1011
para ler estruturas, 421
para ler um arquivo, 404
Read, membro, usando, para ler dados do canal de arquivos, 1005
Readdir, função, usando, para ler um diretório do DOS, 432
ReadFile, função, 1457
ReadFileEx, função, 1495, 1497
Recursão direta e indireta, definição, 244
Recursão,
compreendendo, 240-44
direta, definição, 581
indireta, definição, 244
removendo, 247
Recursiva, função fatorial, compreendendo, 241
Redireção, E/S, evitando, 663
Redirecionamento da E/S, evitando, 663
Redirecionamento da entrada, compreendendo, 652
Redirecionamento da saída, compreendendo, 651
RedrawWindow, função, 1354
Redução da força compreendendo, 786
Referência, passando para uma função, 848
Referenciando a declaração, definição, 794
Referências, regras para trabalhar com, 851
RegisterClass, função, Windows, compreendendo, 1269
RegisterClassEx, função, compreendendo, 1282
Registrador de segmento,
DOS, tabela de, 571
palavra-chave, 797
Registradores,
compreendendo, 551
salvando e armazenando, 737
segmento do DOS, tabela de, 571
Regras de sintaxe, definição, 2
Regras explícitas, definição, 708
Regras implícitas, definição, 5
REGS, união, usando a, 483
para exibir a versão atual do DOS, 484
Reinterpret_cast, operador, usando o, 1150
ReleaseDC, função, usando, 1430
ReleaseSemaphore, função, usando para incrementar o contador de um semáforo, 1412
Remove, função,
classe *linked_list*, compreendendo, 1185
usando, para excluir um arquivo, 377
RemoveDirectory, função, usando para excluir um diretório *Vazio*, 377
RemoveProp, função, usando para remover propriedades, 1283
Rename, função, usando para renomear um arquivo, 376
ReplyMessage, função, usando, 1294
ResetEvent, função, usando para reiniciar o estado de um evento para não-sinalizado, 1413
ResumeThread, função, 1403
Return, comando,
compreendendo, 259, 681
usando 258
Rewinddir, função, 434
RGBQUAD, estrutura, membros, tabela de, 1433
Rmdir, função, usando, para remover um diretório, 397
Rmtmp, função, usando para remover um nome de arquivo temporário, 389
Rotação, bit a bit, efetuando, 95
Rotina de finalização de callback, usando uma, 1498
Rotina de tratamento de erro crítico, criando uma, 772-773
Rotina de tratamento de interrupção,
criando uma, 763, 767
definição, 761
Rotina de tratamento,
exceção,
definição, 1126
escrevendo uma simples, 1128
interrupção,
criando uma, 763, 767
definição, 761
Rotinas de tratamento de exceção,
alto nível, 1393
definição, 1126
escrevendo uma simples, 1128
Rotinas, arquivo de biblioteca, listando, 695
Rótulo, definição, 128
RTTI, compreendendo, 1155
Rvalue, compreendendo, 796

Saída de string, mais rápida usando a função *cputs*, 300
Saída exponencial, formatação, 69
Saída,
exibindo, em uma nova linha, 7
exponencial, formatação, 69
formatada mais rápido, usando a função *cprintf*, 297
largura, controlando, usando o canal de E/S *cout*, 826
printf, justificando à esquerda, 70
string mais rápida, usando a função *cputs* para, 300
tela, garantindo, 658
Scroll_Window, programa, 1352-53
SCROLLBAR, classe, Windows, definição, 1276
ScrollDC, função, usando, 1357

SCROLLINFO, estrutura, membros da, tabela, 1351
ScrollWindowEx, função, usando para controlar a rolagem da janela, 1352
SearchPath, função, usando para procurar um arquivo, 1478
Seção crítica, tipo de sincronização de encadeamento, criando, 1405
Seekg, método, usando para acesso aleatório, 1022, 1033
Seekp, método, usando para acesso aleatório, 1022, 1033
Seleção do conjunto de instruções, aumentando o desempenho com, 777
Seleção, método de classificação, 494, 495
Seleções do usuário, respondendo, dentro de uma caixa de lista, 1333
Semáforos,
  tipo de sincronização do encadeamento, 1404
  usando, 1412
SendMessage, função, compreendendo,
Senha, pedindo uma, 649
Seno hiperbólico, definição, 330
Seqüências de escape, ANSI, compreendendo, 77
Serviços de arquivo, DOS, usando, 410
Serviços de teclado da BIOS, usando o, 562
Serviços da BIOS, compreendendo, 550
Serviços do sistema, compreendendo, 549
Set_new_handler, função, usando para instalar uma rotina de tratamento personalizada, 871
Setbase, modificador, usando, 833
Setbuf, função, usando para atribuir um buffer de arquivo, 418
Setcbrk, função, usando para definir o estado da verificação de Ctrl+Break, 555
SetCurrentDirectory, função, usando para mudar o diretório atual, 1468
Setdate, função, usando, 635
SetDlBits, função, 1438
SetDlBitsToDevice, função, usando para escrever em um determinado dispositivo, 1439
SetDoubleClickTime, função, 1343
Setdta, função, usando para controlar a área de transferência do disco, 561
SetEvent, função, usando para definir o estado de um evento como sinalizado, 1413
Setf, função, sobrecarregada, usando para definir os sinalizadores de formato, 986
SetFileAttributes, função, usando para definir os atributos de um arquivo, 1464
SetFilePointer, função, usando para abrir arquivo, 1455
Setiosflags, manipulador, usando, 831
SetMenu, função, 1302
Setmode, função, usando para especificar o modo de tradução, 407
Setprecision, manipulador, usando, 830
SetPriorityClass, função para definir a classe de prioridade de um processo, 1399, 1401
SetProcessWorkingSetSize, função, usando para definir o Tamanho do *working set*, 1489
SetProp, função, parâmetros para a, 1283
SetScrollInfo, função, usando para definir a posição e o intervalo da barra de rolagem
SetThreadPriority, função, usando para definir o nível de prioridade de um encadeamento, 1400
Settime, função, usando a, 634
SetUnhandledExceptionFilter, função, usando, 1393
Setvbuf, usando para alocar um buffer de arquivo, 419
Setw, manipulador, usando para definir a largura *cout*, 827
SetWindowHookEx, função, usando a, 1296
Shell, método de classificação
  compreendendo o, 496
  usando para classificar uma matriz, 497
Show_2d_array, função, usando, 476
Show_array, função, usando para exibir valores da matriz, 461
Show_directory, função, usando para exibir nomes de arquivo, 435
Show_Windows, função, compreendendo, 1268
ShowScrollBar, função, usando para exibir ou ocultar as barras de rolagem, 1348
Sin, função, usando, 329
Sinal de subtração (-), operador, sobrecarregando, 948
Sinal do valor, exibindo, usando a função *printf*, 64
Sinalizadores,
  CreateProcess, função, 1377
  definindo todos, 988
  formatação *ios*, 983
  formato,
    definindo os, 984
    zerando os, 985
  registrador, compreendendo o, 552
Sinh, função, usando a, 330
Sistema operacional
  criação de encadeamento, passos, 1387
  número de versão, determinando, 726
Sistema operacional monotarefa, definição, 581
Sistema, erro, detectando, 724
Sizeof, operador,
  C, compreendendo, 93
  determinando o número de elementos em uma matriz, 506
  determinando os tamanhos das classes, 1102
  exibindo a memória usada pelas matrizes 456
Sleep, função usando para especificar um intervalo de pausa, 557
Small, modelo de memória, compreendendo o, 615
Sobrecarga de função, *versus* argumentos normais da função, 1143
Sobrecarga de operador, 945
  restrições, 951
  usando funções amigas para, 952, 955
Sobrecarregando,
  compreendendo, 864
  evitando a ambigüidade, 867
  operador,
    restrições, 951
    usando funções amigas para, 952, 955
  programas de exemplo, 865, 866
  uma função genérica, 1114
Sopen, função, usando, para abrir um arquivo para acesso Compartilhado, 426
Sound, função, usando para gerar sons, 558
Spawnl, função, 754
Spawnlpe, funções, usando, 756
Sprintf, função, usando para criar um nome de arquivo, 213
Sqrt, função, 348
Sscanf, função, 214
Starvation, prioridade do encadeamento, 1396
STATIC, classe do Windows, definição, 1276
Static, palavra-chave,
  ao declarar membros da classe, 1073
  para declarar variáveis, 234
  usando a, 269
Static_cast, operador, usando o, 1151
Status da operação de arquivo, verificação, 1006
Stdaux, indicativo de arquivo, usando, 666
Stdin, indicativo de arquivo,
  introdução a, 652
  usando, 654
Stdlib.h, arquivo de cabeçalho, macro min e max, usando, 341
Stdout, indicativo de arquivo
  introdução a, 651
  usando, 654
Stdprn, indicativo de arquivo, usando, 664
Strime, função, usando a, 643
Str_index, função, usando para obter um índice para um caractere, 177
Strand, função, usando, 347
Strcat, função, usando para concatenar strings, 169

Strchr, função, usando para encontrar a primeira ocorrência de um caractere, 176
Strcmp, função, usando, para comparar strings de caracteres, 185
Strcpy, função, usando para copiar caracteres de uma string para outra, 168
Strdup, função, usando para duplicar o conteúdo de uma String, 189
Stream. *Veja* canal
Streql, função, usando para comparar strings, 173
Strfirma, função, usando, 646
Stricmp, função, usando para comparar string, 187
Strieql, função, usando para comparar string, 174
String de data e hora, formatada, criando, 646
String de data, obtendo uma, 627
String de horário, obtendo, 628
Strings de caracteres,
  canais, compreendendo, 1026
  como C armazena, 163
  comparando usando a função *strcmp* para, 185
  convertendo para um valor numérico, 188
  definição, 62
  escrevendo uma,
    usando a função *puts*, 299
    usando *utrstream*, 1027
  exibindo, usando a função *printf*, 62
  lendo
    entrada de, 214
    do teclado,
      usando a função *cgets*, 302
      usando a função *gets*, 301
  matriz de,
    classificando, 501
    percorrendo em um laço, 523
    visualizando uma, 522
  representação do compilador da, 162
Strings *far*, trabalhando com, 180
Strings,
  arquivo, trabalhando com, 180
  atribuindo entrada do teclado a, 289
  C, visualizando, 161
  caixa, convertendo, 175
  caractere,
    como C armazena uma, 163
    comparando, usando a função strcmp, 185
    convertendo para um valor numérico, 188
    definição, 62
    escrevendo, usando a função *puts*, 299
    exibindo, usando a função *printf*, 62
    lendo do teclado,
      usando a função *cgets*, 302
      usando a função *gets*, 301
    lendo entrada do, 214
    matriz de,
      classificando 501
      percorrendo em um laço, 523
      visualizando uma, 522
    representação do compilador, 162
  comprimento, determinando, 166
  concatenação, definição 169
  conteúdo,
    duplicando, usando a função *strdup*, 189
    invertendo, 183
  data e hora formatadas, criando, 646
  definição, 161
  encontrando cada ocorrência, dentro da entrada redirecionada, 667
  far, exibindo, 73
  inicializando uma, 216
  near, exibindo, 73
  número de caracteres em, encontrando, 167
  passando para as funções, 248
  percorrendo em um laço, usando um ponteiro, 518
  sobrescrevendo, usando a função *strset* para, 184
  substring de, localizando usando a função *strstr*, 191
Strlen, função, usando para encontrar o número de caracteres em uma string, 167
Strlwr, função, usando para converter string para minúscula, 175
Strncat, função, usando para concatenar os primeiros *n* caracteres de uma string, 170
Strncmp, função, usando para comparar os primeiros *n* caracteres de uma string, 186
Strncmps, função, usando para comparar strings sem caixa, 187
Strr_index, função, usando para obter um índice para um caractere, 179
Strrchr, função, usando para encontrar a última ocorrência de um caractere, 178
Strrev, função, usando para inverter o conteúdo de uma string, 183
Strset, função, usando para sobrescrever uma string, 184
Strspn, função, usando para encontrar a primeira ocorrência de um compilador de um conjunto, 190
Strstr, função, usando para localizar uma substring dentro de uma string, 191
Strstr_cnt, função, usando para contar ocorrências de substrings, 192
Strstr_rem, usando para remover uma substring de uma string, 196
Strstr_rep, função, usando para substituir uma substring por outra, 197
Strstream, usando, 1032
Struct, palavra-chave, em C++, 873
Strupr, função, convertendo uma string em maiúsculas 175
Strxfrm, função, usando, para transformar uma string em outra, 171
Subclasses, compreendendo, 1081
Subexpressões, eliminando, 787
Substring_index, função, usando, para obter um índice para uma substring, 193
Superclasses, compreendendo, 1081
Swab, função, 592
Swap_values, função, usando a, 272
SwapMouseButton, função, 1344
Switch, comando,
  compreendendo, *break* dentro, 130
  usando,
    o *case* padrão, 131
    para testar múltiplas condições, 129
Sync_with_stdio, função, 979
Systable.c, programa que exibe a tabela de arquivos, 373

Tabela de alocação de arquivo, lendo informações da, 354
Tabela de arquivos do sistema, 372, 373
Tabelas de string, apresentando, *1313*
Tamanho da pilha,
  determinando o atual do programa, 586
  encadeamento, determinando, 1388
Tamanhos do *working set*,
  compreendendo, 1488
  definição, 1489
Tan, função, 331
Tangente hiperbólica, definição, 332
Tangente, triângulo, definição, 331
Tanh, função, usando, 332
Teclado
  entrada, atribuindo a uma string, 289
  eventos, respondendo aos, 1339
  mensagens, janelas, tabela de, 1339
  serviços da *BIOS*, usando, *562*
Teclas virtuais, 1340, 1341
Tee, comando, escrevendo um simples, 858
Tela,
  cor,
    controlando, 313
    exibindo,

com a função *outtext*, 303
com o controlador de dispositivo ANSI, 79
saída, executando rápida, usando a função *putch*, 295
tela, limpando,
com a função *clrscr*, 304
com o controlador de dispositivo ANSI, 78
texto, movendo, 319
Tell, função, usando para determinar a posição do ponteiro de arquivo, 436
Tellg, método para localizar a posição do ponteiro, 1020, 1023
Tellp, método para localizar a posição atual do ponteiro, 1020
TEMP, diretório de arquivos, abrindo, 392
Tempnam função, usando para criar um nome de arquivo temporário, 387
Tempo de processamento, determinando, 625
Temporizador da BIOS, lendo o, 629
Temporizador do PC, interceptando, 770
TerminateThread, usando para finalizar um encadeamento, 1394
Textattr, função, controlando a saída de cores na tela, 313
Textbackground, função, definindo a cor do segundo plano, 316
Textcolor, função, definindo a cor do primeiro plano, 315
Textcopy.cpp, compreendendo, 446
Textmode, usando para determinar o modo de texto atual, 318
Texto
escrevendo linhas em um arquivo, 444
intensidade, controlando, 317
janela, definindo uma, 320
lendo linhas em um arquivo, 443
modo, definições,
determinando o atual, 318
determinando usando *gettextinfo*, 312
tela, movendo, 319
This, ponteiro, compreendendo, 1084, 1085
Throw, comando, compreendendo, 1129
Time, função, 621
Timezone, função, 637
Tiny, modelo de memória, compreendendo, 614
Tipo de caractere,
DWORD, compreendendo, 1275
LPCTSTR, compreendendo, 1274
Tipo de dados
bool, apresentando, 1161
C, definição, 37
Tipo enumerado,
atribuindo um valor a, 736
usando, 733, 734
Tipo,
enumerado, atribuindo um valor a, 736
string,
características, definindo as, 1164
criando uma, 1163
variável,
compreendendo, 25
definição, 23
suportada por C, 25
Tipos de arquivo, compreendendo, 11
Tipos de indicativos do Windows, tabela, 1260
Tipos de relógio do PC, compreendendo, 647
Tipos derivados versus tipos fundamentais, 790
Tipos fundamentais, *versus* tipos derivados, 790
Tipos,
básicos de C, 789
criando seus próprios, 48
dados, C, definição, 37
derivados, tipos fundamentais, 790
enumerados, usando, 733, 734
fundamentais versus derivados, 790
Tmpnam, função para criar nome de arquivo temporário, 386
Toascii, macro do arquivo de cabeçalho ctype.h, 212
Tokenizar, definição, 215
Tokenize_string função, 215
Tolower, função, compreendendo, 211
Toupper, função, compreendendo, 210
Traduções de arquivo, compreendendo, 366
Tratamento da exceção,
aplicando, 1140
compreendendo, 1126
forma básica, 1127
Try, bloco,
localizando para uma função, 1132
pegando todas as exceções com um único, 1136
usando para tratamento de exceção, 1127-1140
TryEnterCriticalSection, função, 1406
Type_info, classe, compreendendo, 1157
Typeid, operador,
usando, para identificação de tipo de execução, 1156
valores de retorno, 1157
Tzset, função, 637

UExitCode, parâmetro, *ExitProcess* função, usando para especificar códigos de saída do processo, 1378
Umask, função, usando para controlar como um arquivo abre, 417
Ungetc, função, usando para "anular a leitura" de um caractere, 448
Ungetch, função, usando para "anular a leitura" de um caractere, 296
Unidade Central de Processamento (CPU), definição, 551
Unidade de disco
determinando a atual, 350
selecionando a atual, 351
Unidade de disquete, testando se está preparada, 358
Uniões
anônimas,
introdução, 918
usando em C++, 823
C++, compreendendo, 917
compreendendo, 481
Unlink, função, usando para excluir um arquivo, 377
Unsigned int, valores do tipo, exibindo com *printf*, 56
UpdateWindow, função, 1354
Usando a BIOS para acessar a impressora, 554
Usando para alocar memória de um heap para processos específicos, 139
Using, comando, usando o, 1154
Utime função, usando para definir a data e a hora de um arquivo, 414

Va_arg, *va_end*, *va_start*, macros, compreendendo, 282, 283
Validações do heap, definição, 603
Valor de retorno, definição, 219
Valor enumerado, compreendendo, 735
Valor hexadecimal, atribuindo um, 49
Valor inteiro, formatando, usando printf para, 65
Valor octal, atribuindo um, 49
Valores de ponto flutuante,
definição, 36
exibindo, usando a função printf, 60
formatando, usando printf para, 68
quebrando em componentes, 342
valor absoluto, determinando, 335
Valores de status de saída, usando para desligar um processo, 1377
Valores,
absoluto, calculando, 325
atribuindo,
a variáveis, na declaração, 28
para uma variável, 24
combinando diferentes, com o canal de E/S *cout*, 808
enumerados, compreendendo, 735
escrevendo, com o canal de E/S *cout*, 807
grandes, trabalhando com, 41
hexadecimal, atribuindo um, 49

inteiro, formatando, usando *printf*, 65
modificáveis e não-modificáveis, 795
octal, atribuindo, 49
parâmetro, fornecendo o padrão, 825
ponteiro, usando, 513
ponto flutuante,
dividindo em componentes, 342
formatando, usando *printf*, 68
retornando um, a partir de uma função, 258
tipo,
char, exibindo com *printf*, 59
float, exibindo com *printf*, 58
int, exibindo com *printf*, 54
long int, exibindo com *printf*, 57
unsigned int, exibindo com *printf*, 56
Variáveis estáticas, como C inicializa, 235
Variáveis locais, compreendendo, 1359
Variáveis locais, definição, 221
Variáveis,
atribuindo um valor ao, 24
na declaração, 28
comentando na declaração, 27
compreendendo, 23
declarando, 23
onde necessário, 834
usando a palavra-chave *static*, 234
definição, 507
endereço,
determinando, 229, 508
usando um, 230
escrevendo, com o canal de E/S *cout*, 807
estáticas, como C inicializa, 235
externas, compreendendo, 267
globais,
_psp, usando, 731
definição, 222
directvideo, 723
evitando, 223
locais,
compreendendo, 218
definição, 221
múltiplas,
declarações, 26
inicializando na declaração, 29
nome,
conflitos, resolvendo, 224
usando significativos, 30
ponteiro, declarando um, 230, 511
tipo,
char, compreendendo, 33
compreendendo, 25
definição, 23
double, compreendendo, 35
float, compreendendo, 34
int, compreendendo, 32
suportada por C, 25
usando dentro de funções, 253
Variáveis-membro, compreendendo, 884
Variável global,
_psp, usando o, 731
definição, 222
directvideo, 723
environ, usando, 685
escopo, definindo, 225
evitando, 223
Versão, exibindo, usando a união REGS para, 484
Vetor de *interrupção*,
definição, 761
definindo um, 765
determinando um 764
Vetor, interrupção,
definição, 761
definindo um, 765
determinando um, 764
Vídeo, direto, controlando, 723
Viewport, origem do, definição, 1418
Virtual, palavra-chave, usando, 1069
VirtualAlloc, função, 1371
VirtualFree, função, usando para liberar memória, 1374
VirtualQuery, função, 1375
Visibilidade, identificador, definição, 279
Visualização do arquivo, definição, 1459
Void, palavra-chave, usando, 275, 678
Volatile, palavra-chave, compreendendo, 270

WaitForMultipleObject, função, usando,
com processamento assíncrono, 1494
para sincronizar muitos encadeamentos, 1409
WaitForSingleObject, função, 1408
WArrows, parâmetro da função *EnableScrollBar*, 1356
Wherex, função para determinar a posição do cursor, 308
Wherey, função para determinar a posição do cursor, 308
Win32 API, 1251
WIN32_FIND_DATA, estrutura, 1474
Window, função, usando para definir uma janela de texto, 321
WindowFromDC, função, usando, 1431
Windows 95, compreendendo o modelo de memória, 1358
Windows NT, modelo de memória, compreendendo, 1358
Windows,
classe *BUTTON*, definição, 1276
classe *COMBOBOX*, definição, 1276
classe *EDIT*, definição, 1276
classe *LISTBOX*, definição, 1276
classes predefinidas,
compreendendo, 1276
usando, para criar uma janela simples, 1277
SCROLLBAR, classe, definição, 1276
STATIC, classe, definição, 1276
Windows, programa,
componentes do, compreendendo, 1273
genérico, criando, 1257
versus programas não-Windows, 1252
WinMain, função, compreendendo, 1265
WndProc, função, 1278
Write, função, usando,
para escrever dados binários, 1012
para escrever em um arquivo, 404
usando, para gravar estruturas, 421
Write, membro, *usando para escrever dados no canal de arquivo*, 1005
WriteFile, função, usando para escrever dados no arquivo, 1456
WriteFileEx, função, usando para enviar solicitações de processamento para a fila APC, 1495
Ws, manipulador, usando para descartar o espaço em branco preliminar, 966

XCOPY, comando do DOS, compreendendo, 680

Zona horária,
obtendo informações, 642
definindo, com a função *tzset*, 639

PROGRAMANDO EM C/C++

"A Bíblia"

Crie programas C++ hoje mesmo, usando o compilador Borland C++ Lite incluído no CD-ROM que acompanha este livro

- C/C++ **Crie programas C/C++ usando o compilador Borland Turbo C++ Lite – incluído no CD-ROM**
- C/C++ **Domine a linguagem de programação C e migre sem dificuldade para C++**
- C/C++ **Aprenda a sobrecarregar funções fornecendo valores de parâmetros padrões e usando gabaritos para reduzir a codificação e ganhar tempo**
- C/C++ **Utilize as exceções de C++ para criar programas poderosos que respondam aos erros incomuns**
- C/C++ **Domine a programação para Windows, que inclui caixas de diálogo, arquivos de recurso, processamento de mensagens e muito mais**
- C/C++ **Use as classes da Biblioteca Standard Template para simplificar os programas complexos**
- C/C++ **Utilize os encadeamentos com base no Windows e a E/S assíncrona e depois avance para a E/S do mouse, portas e canais nomeados**
- C/C++ ***Programando em C/C++ — A Bíblia*, do Jamsa, é realmente o melhor guia de programação em C/C++ que você pode ter**

O CD contém o código-fonte do livro e o compilador Borland C++ Lite

ISBN 85-346-1025-8

MAKRON Books

Visite o nosso site
http://www.makron.com.br

JAMSA PRESS

CÓPIA MF0F0T010